MEMORIA

Alberto Pérez Hervella
Laura Regojo Barrón
Uxía Pousa Frasquet

ÍNDICE

1.Supermercado..... 2
1.1 Compra del robot al supermercado y gestión de las cervezas favoritas.....2
1.2. Compras (supermercado a proveedor). Precios de los productos aleatorios y actualizables.....10
1.3 Generar dinero aleatoriamente.....11
2. Owners..... 11
2.1. Colaboración/cooperación solicitud de cervezas y pinchos..... 11
3. Myrobot..... 14
3.1. Servir cervezas y pinchos al owner.....14
3.2. Recoger los platos sucios del owner y llevarlos al lavavajillas.....18
3.3. Llevar los platos limpios a la alacena.....21
4. Cleaner..... 22
4.1. Recoger las latas del grid, colaboración entre Owner y myCleaner..... 22
4.2. Llevar las latas a la papelera.....25
5. Burner..... 26
5.1. Vaciar el contenido de la papelera cuando se llene.....26
6. Obstáculos.....28
6.1. Generación aleatoria de obstáculos.....28
6.2. Los agentes deben esquiva Obstáculos generados, Otros agentes y Owners Nevera/alacena/lavavajillas/papelera.....29
7. Lógica de movimiento..... 31
7.1. Posicionado en casillas adyacentes y movimiento no diagonal.....31

1.Supermercado

1.1 Compra del robot al supermercado y gestión de las cervezas favoritas

En primer lugar, tendremos en cuenta que tanto el supermercado como la nevera y el resto de elementos que gestionan las compras al supermercado van a trabajar con distintas marcas de cerveza y múltiples tipos de pincho.
En este apartado vamos a usar los agentes myRobot, myRobotDel y mySupermarket.
Primero, nos situamos en el fichero myRobot.asl, donde el siguiente plan es importante, porque gracias a él nos aseguramos de que la nevera nunca quede vacía. Para ello, cada vez que myRobot vaya a la nevera comprueba el número de cervezas y pinchos que hay; si hay menos de 6, myRobot envía un goal a myRobobtDel para realizar el plan de comprar (*buy)*.

```
+!take(Ag,fridge, P, Brand, Pincho) : not too_much(beer)<-
    .println("El robot está cogiendo una cerveza y un pincho para acompañar.");
    .send(myFridge, askOne, stock(P, Brand, S), stock(P, Brand, S));
    .send(myFridge, askOne, stock(pincho, Pincho, Sp), stock(pincho, Pincho, Sp));
    if(S < 6 | Sp < 6){
        .send(myRobotDel, achieve, buy(P, Brand));
        .send(myRobotDel, achieve, buy(pincho, Pincho));
    }
    !check(Ag,fridge, P, Brand,Pincho).
```

El agente myRobotDel se va a encargar de realizar las compras al supermercado.
Para la compra de las cervezas hay 3 escenarios posibles, los cuáles vamos a manejar en una misma función *!buy(beer, Brand)*.

```
+!buy(beer, Brand): wallet(M) & tamPackBeer(Qtd)<-
    !comparePrices(beer,Brand);
    ?cheaper(Super, beer, Brand, Stock, Precio);
        if(Qtd <= Stock){
            //Escenario 1
            !order(Super,beer,Brand, Qtd);
        }else{
            //Escenario 2, caso B
            Resto = Qtd - Stock;
            ?barata(Super, beer, BrandB, StockB, PrecioB);
            if(Stock > 0){
                !orderDosTipos(Super, beer, Brand, Stock, BrandB, Resto);
            }else{
            //Escenario 3
                ?masbarataAll(SuperC, beer, BrandC, StockC, PrecioC);
                if(StockC >= Qtd){
                    !order(SuperC, beer, BrandC);
                }else{
                    .println("No he podido comprar más Cervezas, no tienen Stock Suficiente");
                }
            }
        }.
```

Antes de nada, para manejar todos estos escenarios vamos a tener que recaudar toda la información sobre los productos de todos los supermercados. Para ello, tenemos unos planes iniciales para que nada más ejecutar el programa tengamos toda la información.

En primer lugar, mediante el plan *!sendPrices(Type, Brand)*, vamos a pedir la información a cada supermercado sobre un producto concreto, que llegará en forma de la siguiente creencia *sprice(mySupermarket,Product,Brand,St1,P1)*, siendo *Supermarket*, el nombre del supermercado, *Product*, el tipo de producto, *Brand*, la marca del producto, *St1*, el stock de ese producto en ese supermercado y *P1* el precio de ese producto en ese supermercado. Luego, vamos a manejar esos datos que nos han llegado con el plan *!pillarDatos.* Con ese plan vamos a ser capaces de sacar la cerveza más barata de cada uno de los supermercados y además la cerveza más barata ofrecida por todos los supermercados combinados. Esto será información muy útil para después manejar los 3 escenarios.

```
!sendPrices(beer, mahou).
!sendPrices(beer, heineken).
!sendPrices(beer, estrella).
!sendPrices(pincho, tortilla).
!sendPrices(pincho, empanada).
!pillarDatos.

+!sendPrices(Product, Brand)<-
    .send(mySupermarket,achieve,send_price(Product, Brand));
    .send(mySupermarket2,achieve,send_price(Product, Brand));
    .send(mySupermarket3,achieve,send_price(Product, Brand)).

+!pillarDatos<-
    .wait(500);
    !cheapestBeer(mySupermarket);
    !cheapestBeer(mySupermarket2);
    !cheapestBeer(mySupermarket3);
    !cheapestBeerAll.

+!cheapestBeer(S)<-
    .findall(P,sprice(S,beer,_,_,P),L);
    .min(L,PrecioBarato);
    ?sprice(S,beer,Brand,Stock,PrecioBarato);
    +barata(S, beer, Brand, Stock, PrecioBarato).

+!cheapestBeerAll<-
    .findall(P,sprice(_,beer,_,_,P),L);
    .min(L,PrecioBarato);
    ?sprice(S,beer,Brand,Stock,PrecioBarato);
    +masbarataAll(S, beer, Brand, Stock, PrecioBarato).
```

Una vez recogidos estos datos, pasamos a *!buy(beer, Brand)*. Primeramente, al entrar en el plan llamamos al plan *!comparePrices(beer,Brand)*, que nos va a permitir saber cuál es el

supermercado que ofrece el producto y marca ,que queremos, más barato, lo cual nos lo dará de la siguiente manera: *cheaper(Super, beer, Brand, Stock, Precio)*.

```
+!comparePrices(Product,Brand)<-
        .wait(100);
        ?sprice(mySupermarket,Product,Brand,St1,P1);
        ?sprice(mySupermarket2,Product,Brand,St2,P2);
        ?sprice(mySupermarket3,Product,Brand,St3,P3);
        if(P1 <= P2 & P1 <= P3) {
            +cheaper(mySupermarket,Product,Brand,St1,P1);
        }
        elif (P2 <= P1 & P2 <= P3) {
            +cheaper(mySupermarket2,Product,Brand,St2,P2);
        }
        elif (P3 <= P1 & P3 <= P2) {
            +cheaper(mySupermarket3,Product,Brand,St3,P3);
        }.
```

El escenario 1 lo manejamos en el primer if, *if(Qtd <= Stock)*. Qtd corresponde con la cantidad que vamos a pedir y Stock con el stock disponible del producto que vamos a comprar en el supermercado ya elegido. Si Qtd es menor que stock, llamamos al plan *!order(Super,beer,Brand, Qtd)* que se va a encargar del pedido del producto al supermercado.

```
if(Qtd <= Stock){
    //Escenario 1
    !order(Super,beer,Brand, Qtd);
```

Si no sucede esto, entramos en el else, y ahí manejaremos el escenario 2, concretamente el caso B. Aquí, dado que ya sabemos que no hay stock suficiente de ese producto en el supermercado para realizar la compra, vamos a completarla con la cerveza más barata de ese mismo supermercado, cuya información nos la facilita la creencia *?barata(Super, beer, BrandB, StockB, PrecioB)*. Sin embargo, antes de comprar ya los dos tipos de cervezas en el supermercado, comprobamos mediante un if, *if(Stock > 0)*, si el stock de la cerveza que queremos comprar es mayor que 0. Si se cumple compramos los dos tipos de cervezas del supermercado y completamos la compra llamando al plan *!orderDosTipos(Super, beer, Brand, Stock, BrandB, Resto),* si el Stock no cumple la condición pasamos al escenario 3.

```
}else{
    //Escenario 2, caso B
    Resto = Qtd - Stock;
    ?barata(Super, beer, BrandB, StockB, PrecioB);
    if(Stock > 0){
        !orderDosTipos(Super, beer, Brand, Stock, BrandB, Resto);
```

Si el stock es 0, estaríamos entonces en el escenario 3, es decir, compraremos la cerveza más barata de todas las ofrecidas por los supermercados. Esta información ya la tenemos recogida, como se explicó anteriormente, en la creencia *masbarataAll(SuperC, beer, BrandC, StockC, PrecioC)*, y de nuevo, comprobamos que el stock de esta cerveza, StockC, sea mayor que la cantidad requerida. Si es así, *order(SuperC, beer, BrandC),* en caso contrario, mandamos un mensaje por pantalla.

```
}else{
//Escenario 3
    ?masbarataAll(SuperC, beer, BrandC, StockC, PrecioC);
    if(StockC >= Qtd){
        !order(SuperC, beer, BrandC);
    }else{
        .println("No he podido comprar más Cervezas, no tienen Stock Suficiente");
    }
```

Para la compra de pinchos vamos a utilizar el siguiente plan, *!buy(pincho, Brand).*

```
+!buy(pincho, Brand)<-
    .wait(100);
    !comparePrices(pincho,Brand);
    ?cheaper(S,pincho,Brand,Stock,_);
    -cheaper(S,pincho,Brand,Stock,_);
    !order(S,pincho,Brand,1).
```

Primeramente, empleamos el plan *!comparePrices(pincho,Brand)* para obtener el supermercado que ofrece el producto solicitado más barato. Después, realizamos el plan *!order(S,pincho,Brand,1*) para pedir 1 artículo del tipo de producto, ya que más adelante partiremos ese producto en porciones y haremos el pincho con el plato.

Ahora, pasamos a explicar cómo se piden las cervezas al supermercado, una vez ya decidido qué se va a pedir.
Las peticiones al supermercado se realizan mediante el siguiente plan, mencionado anteriormente, *!order(Supermarket,Product,Brand,Qtd)*, que se ejecutará si no hay ya un "order" en camino, *not ordered(Product)*, y si ya han acabado de entregarse productos anteriormente pedidos, *not deliveredP & not deliveredB*. De esta forma, evitamos problemas de concurrencia.

```
+!order(Supermarket,Product,Brand,Qtd) : not ordered(Product) & not deliveredP & not deliveredB<-
    .println("El robot ha realizado un pedido al supermercado.");
    !go_at(myRobotDel,delivery);
    .println("El robot va a la ZONA de ENTREGA.");
    ?wallet(M);
    .send(Supermarket, achieve, order(Product,Brand,Qtd,M));
    +ordered(Product).
```

En este plan, el robot se dirige hacia la zona delivery mediante el plan *go_at(myRobotDel,delivery)*, y una vez llegado le enviamos la información al supermercado correspondiente para que complete la compra, *order(Product,Brand,Qtd,M)).*
Para las cervezas, nos faltaría explicar el caso especial, el caso B del escenario 2, que como comentábamos anteriormente utilizamos el plan, *!orderDosTipos(Super, beer, Brand, Qtd, BrandB, Resto)*, y la mecánica es igual que con un pedido de cervezas normal, salvo que, en este caso, le enviamos información del segundo tipo de cerveza que se va a comprar, *orderMulti(beer,Brand, Qtd ,BrandB, Resto,M)*.

```
+!orderDosTipos(Super, beer, Brand, Qtd, BrandB, Resto)<-
    .println("El robot ha realizado un pedido múltiple al supermercado.");
    !go_at(myRobotDel,delivery);
    .println("El robot va a la ZONA de ENTREGA.");
    ?wallet(M);
    .send(Supermarket, achieve, orderMulti(beer,Brand, Qtd ,BrandB, Resto,M ));
    +ordered(beer).
```

Para las peticiones de pinchos realizamos exactamente lo mismo que la petición “normal” de una cerveza. El robot se dirige hacia la zona delivery mediante el plan *go_at(myRobotDel,delivery)*, y una vez llegado le enviamos la información al supermercado correspondiente para que complete la compra, *order(pincho,Brand,Qtd,M))*.

```
+!order(Supermarket,pincho,Brand,Qtd) : not ordered(pincho) & not deliveredP & not deliveredB<-
    .println("El robot ha realizado un pedido al supermercado.");
    !go_at(myRobotDel,delivery);
    .println("El robot va a la ZONA de ENTREGA.");
    ?wallet(M);
    .send(Supermarket, achieve, order(pincho,Brand,Qtd,M));
    +ordered(pincho).
```

Ahora pasamos al fichero mySupermarket.asl para realizar la compra. Empezaremos con las cervezas.
Como se comenta anteriormente, la información fue enviada de myRobotDel mediante un achieve del plan *order(Product, Brand, Qtd, M)*. El plan *!order* tiene varios casos dependiendo de si myRobotDel tiene suficiente dinero para comprar o no. Si no tiene suficiente, se avisará al agente myRobotDel para que el Owner le mande dinero, *notEnoughM(Qtd, Price)*, y ya con la wallet de myRobotDel actualizada se volvería a llamar al plan.

```
+!order(Product, Brand, Qtd,M)[source(Ag)]: stock(Product, Brand, A, Price) & Qtd*Price > M<-
    ?stock(Product, Brand,_,Price);
    .println(Ag, " no tiene suficiente dinero para comprar");
    .send(Ag, achieve, notEnoughM(Qtd, Price));
    .send(Ag, askOne, wallet(N));
    !order(Product, Brand, Qtd, N)[source(Ag)].
```

Si el Owner sí que tuviera suficiente dinero para realizar la compra, se actualizan los stocks del supermercado, se añaden las cervezas al fridge mediante la acción *deliver(Product, Qtd),* se manda un plan a myRobotDel para que actualice su wallet, *delivered(Product, Brand, Qtd,OrderId, Qtd*Price),* y también se actualizará el wallet del supermercado con el correspondiente ingreso.

```
+!order(Product, Brand, Qtd, M)[source(Ag)]: stock(Product, Brand, A, Price) & A >= Qtd & Qtd*Price <= M <-
    -stock(Product, Brand,_,Price); //elimino la creencia de stock del producto (ya que si tengo otro product
    +stock(Product, Brand,A-Qtd,Price); //añado la creencia ya actualizada
    ?stock(Product, Brand,X,Price);
    .println(" tengo un stock de ", X, Product, Brand);
    .println(" cada ",Brand," cuesta ", Price);
    deliver(Product, Qtd);
    .send(Ag, tell, deliveredB);
    .send(Ag,achieve, delivered(Product, Brand, Qtd,OrderId, Qtd*Price));
    +orderFrom(Ag, Qtd);
    .println("Pedido de ", Qtd," ",Brand, " recibido de ", Ag);
    -wallet(Pro);
    +wallet(Pro+Qtd*Price);
    ?wallet(Y);
    .println("el saldo actual es de", Y).
```

La acción *deliver(Product, Qtd)* va a ser interpretada en MyHouseEnv.java y se realizará la función *addBeer(int qtd)*, siendo qtd la cantidad de cervezas que se van añadir.

```
} else if (action.getFunctor().equals("deliver") & ag.equals("mySupermarket")) {
    try {
        Thread.sleep(4000);
        result = model.addBeer( (int)((NumberTerm)action.getTerm(1)).solve());
    } catch (Exception e) {
        logger.info("Failed to execute action deliver!"+e);
    }
```

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

Ahora pasamos al caso B del escenario 2, que tal y como comentamos anteriormente, la compra se realizará mediante el plan *!orderMulti(Product,Brand, Qtd ,BrandB, QtdB,M)*.

```
+!orderMulti(Product,Brand, Qtd ,BrandB, QtdB,M )[source(Ag)]:stock(Product, Brand, St, _) & stock(Product, BrandB, StB, _)<-
    -stock(Product, Brand,_,Price);
    +stock(Product, Brand, St -Qtd, Price);
    -stock(Product, BrandB,_,PriceB);
    +stock(Product, BrandB, StB -QtdB, PriceB);
    .println("Pedido de ",Qtd," ",Brand," y ",QtdB," ",BrandB," realizadonse.");
    deliver(Product, Qtd+QtdB);
    .send(Ag, tell, deliveredB);
    .send(Ag, achieve, deliveredDos(Product, Brand, Qtd,BrandB,QtdB,OrderId, Qtd*Price+QtdB*PriceB));
    -wallet(Pro);
    +wallet(Pro+Qtd*Price+QtdB*PriceB);
    ?wallet(Y);
    .println("el saldo actual es de", Y).
```

Este plan es similar al anterior. Actualizamos los stocks de los productos, se añaden las cervezas al fridge mediante la acción *deliver(Product, Qtd+QtdB),* se manda un plan a myRobotDel para que actualice su wallet, *deliveredDos(Product, Brand, Qtd,BrandB,QtdB,OrderId, Qtd*Price+QtdB*PriceB),* y también se actualizará el wallet del supermercado con el correspondiente ingreso.

Ahora, de vuelta con los pinchos, lo cual será prácticamente igual que con las cervezas, salvo que en esta ocasión, la adición de los pinchos en el fridge va a ser diferente, ya que hay que construirlos antes.

```
+!order(pincho, Brand, Qtd, M)[source(Ag)]: stock(pincho, Brand, A, Price) &  A >= Qtd & Qtd*Price <= M <-
    -stock(pincho, Brand,_,Price); //elimino la creencia de stock del producto  (ya que si tengo otro produ
    +stock(pincho, Brand,A-Qtd,Price); //añado la creencia ya actualizada
    ?stock(pincho, Brand,X,Price);
    .println(" tengo un stock de ", X, pincho, Brand);
    .println(" cada ",Brand," cuesta ", Price);
     .send(Ag, tell, deliveredP);
    .send(Ag,achieve, delivered(pincho, Brand, Qtd,OrderId, Qtd*Price));
    +orderFrom(Ag, Qtd);
    .println("Pedido de ", Qtd," ",Brand, " recibido de ", Ag);
    -wallet(Pro);
    +wallet(Pro+Qtd*Price);
    ?wallet(Y);
    .println("el saldo actual es de", Y).
```

También, lo mismo que la cerveza para cuando se quiera comprar un pincho y myRobotDel no tenga dinero suficiente.

```
+!order(Product, Brand, Qtd,M)[source(Ag)]: stock(Product, Brand, A, Price) & Qtd*Price > M<-
    ?stock(Product, Brand,_,Price);
    .println(Ag, " no tiene suficiente dinero para comprar");
    .send(Ag, achieve, notEnoughM(Qtd, Price));
    .send(Ag, askOne, wallet(N));
    !order(Product, Brand, Qtd, N)[source(Ag)].
```

Por último, un caso especial que sólo se va a dar en la compra de pinchos, ya que, en el caso de las cervezas se controla previamente en myRobotDel.asl. Este caso especial sucede cuando el supermercado no tiene suficiente stock para el pedido. El supermercado pedirá a myProvider nuevo stock del producto mediante el envío de un achieve del plan *supply(mySupermarket, Product, Brand, Qtd*2)*.

```
+!order(Product, Brand, Qtd,M)[source(Ag)]: stock(Product, Brand, A, Price) &  A < Qtd <-
    ?stock(Product, Brand,X,Price);
    //.send(myRobot, tell, no_stock_super(Product, Brand,X));
    .print("Sólo me quedan", X , " ", Product," ",Brand);
    .print("Voy hacer restock de ", Product," ",Brand);
    .send(myProvider, achieve, supply(mySupermarket, Product, Brand, Qtd*2 ));
    //.wait(3000);
    !order(Product, Brand, Qtd, M)[source(Ag)].
```

Ahora, ya con todo controlado en el supermercado, volvemos a myRobotDel.asl para realizar los planes !*delivered*, mencionados anteriormente, y ya finalizar con las compras definitivamente.

De nuevo, empezamos con las cervezas. Como mencionamos, el supermercado nos mandó un achieve para el plan *!delivered*. En este plan se va actualizar el wallet del agente myRobotDel y se va a quitar la creencia *ordered(Product)*. Ya que estamos finalizando el pedido, le ordenamos a myFridge que aumente su stock, *addStock(Product, Brand, Qtd)*, y por último, quitamos la creencia *deliveredB*.

```
+!delivered(Product,Brand,Qtd,OrderId, Cost) <-
    -wallet(M);
    +wallet(M-Cost);
    ?wallet(X);
    .println(" ahora me queda: ", X);
    -ordered(Product);
    .send(myFridge, achieve, addStock(Product, Brand, Qtd));
    .abolish(deliveredB);
    .wait(100).
```

Para el caso B del escenario 2, lo mismo pero aplicado a dos productos.

```
+!deliveredDos(Product,Brand,Qtd,BrandB,QtdB,OrderId, Cost) <-
    -wallet(M);
    +wallet(M-Cost);
    ?wallet(X);
    .println(" ahora me queda: ", X);
    -ordered(Product);
    .send(myFridge, achieve, addStock(Product, Brand, Qtd));
    .send(myFridge, achieve, addStock(Product, BrandB, QtdB));
    .abolish(deliveredB);
    .wait(100).
```

Ahora, vamos con los pinchos, que son un caso especial, ya que vamos a tener que construirlos.

```
+!delivered(pincho,Brand,Qtd,OrderId, Cost) <-
    -wallet(M);
    +wallet(M-Cost);
    ?wallet(X);
    .println(" ahora me queda: ", X);
    -ordered(pincho);
    .println("Vamos a coger ",Qtd*5,  " platos para hacer ",Qtd*5," pinchos");
    !go_at(myRobotDel,cboard);
    takeDishes(dishes, Qtd*5);
    .println("Vamos a cortar los/las ",Qtd," ",Brand," para hacer 5 pinchos con cada uno/a");
    !go_at(myRobotDel,fridge);
    deliverP(pincho, Qtd*5);
    .send(myFridge, achieve, addStock(pincho, Brand, Qtd*5));
    //.wait(20);
    !go_at(myRobotDel,base);
    .abolish(deliveredP).
```

Para los pinchos, el plan *!delivered* es diferente; actualizamos el wallet, quitamos la creencia *ordered(pincho)*, dirigimos al agente myRobotDel al cboard (alacena) con el plan *!go_at(myRobotDel,cboard)* para recoger los platos que se utilzarán para construir los pinchos. Cogemos los platos mediante la acción *takeDishes(dishes, Qtd*5)* en la cual multiplicamos por 5 porque vamos a dividir cada producto en 5 pinchos, que va ser interpretada en MyHouseEnv.java y se realizará *removeCB(int n),* donde *n* será el número de platos que se van a utilizar.

```
else if(action.getFunctor().equals("takeDishes") & ag.equals("myRobotDel")){
   try {
       //Thread.sleep(4000);
       result = model.removeCB( (int)((NumberTerm)action.getTerm(1)).solve());
   } catch (Exception e) {
       logger.info("Failed to execute action deliver!"+e);
   }
```

```
boolean removeCB(int n){
    dishCountCB -= n;
    if (view != null)
        view.update(lCBoard.x,lCBoard.y);
    return true;
}
```

Después de coger los platos dirigimos al robot al fridge *!go_at(myRobotDel,fridge)* y ya por fin, añadimos los pinchos construidos a la nevera, mediante la acción *deliverP(pincho, Qtd*5)*, que va ser interpretada en MyHouseEnv.java y se realizará *addPincho(int n),* donde *n* será el número de pinchos que se van a añadir a la nevera.

```
else if (action.getFunctor().equals("deliverP") & ag.equals("mySupermarket")) {
   try {
       Thread.sleep(4000);
       result = model.addPincho( (int)((NumberTerm)action.getTerm(1)).solve());
   } catch (Exception e) {
       logger.info("Failed to execute action deliver!"+e);
   }
```

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

Para finalizar, mandamos a myRobotDel de vuelta a su base con *go_at(myRobotDel,base)* y quitamos la creencia mediante *abolish(deliveredP)*.

1.2. Compras (supermercado a proveedor). Precios de los productos aleatorios y actualizables.

Ahora vamos a ver la comunicación entre el supermercado y el proveedor. Hay que tener en cuenta que hay varios supermercados pero para la explicación nos centraremos en uno sólo, mySupermarket.asl ya que todos funcionarán de la misma manera.
En el archivo mySupermarket.asl tenemos el plan *+!order,* que se ejecuta cuando su stock del par Producto, Marca, (*A*) es menor a la cantidad que le está solicitando el Owner (*Qtd*). Después de lanzar los mensajes oportunos comunicando cuál es su stock actual y su intención de hacer restock, envía un goal al Proveedor para que ejecute el plan *supply(mySupermarket, Product, Brand, Qtd*2)* pidiendole así el doble de la cantidad que el Owner le ha solicitado (*Qtd*2)*.

```
+!order(Product, Brand, Qtd,M)[source(Ag)]: stock(Product, Brand, A, Price) &  A < Qtd <-
    ?stock(Product, Brand,X,Price);
    .print("Sólo me quedan", X , " ", Product," ",Brand);
    .print("Voy hacer restock de ", Product," ",Brand);
    .send(myProvider, achieve, supply(mySupermarket, Product, Brand, Qtd*2 ));
    !order(Product, Brand, Qtd, M)[source(Ag)].
```

En el plan *+!supply* de myProvider.asl, le pasamos el agente que solicita (*Ag)*, el producto (*Product)*, la marca (*Brand)* y la cantidad solicitada (*Qtd)*.
Dentro del plan se elimina la creencia del precio asociado al par Producto, Marca solicitado *.abolish(cost(Product, Brand, _))* y añadimos una nueva con un precio aleatorio dentro de un rango *+cost(Product, Brand,math.round(math.random(4))+1)*. De esta forma, los precios son **aleatorios y actualizables.**
Por último, el proveedor envía al supermercado el goal de ejecutar el plan *restock(Product, Brand, Qtd,Qtd*N))* donde *Qtd* sigue siendo la cantidad de producto solicitada y *Qtd*N* el precio final del pedido.

```
+!supply(Ag, Product, Brand, Qtd )<-
    .abolish(cost(Product, Brand, _));
    +cost(Product, Brand,math.round(math.random(4))+1);
    .wait(100);
    ?cost(Product, Brand, N);
    .send(Ag, achieve, restock(Product, Brand, Qtd,Qtd*N)).
```

En el plan *!restock* del supermercado se actualiza el *stock* y el dinero del supermercado (*wallet)*.

```
+!restock(Product, Brand, Qtd,Cost)<-
    -stock(Product, Brand,X,Price);
    +stock(Product, Brand,X+Qtd,Price);
    -wallet(Pro);
    +wallet(Pro-Cost);
    ?stock(Product, Brand,Y,Price);
    ?wallet(Z);
    .println("Ahora el stock de ",Product, Brand," es de ", Y);
    .println("Ahora el saldo es de ", Z).
```

1.3 Generar dinero aleatoriamente

En los supermercados añadimos esta creencia, para generar un saldo inicial aleatorio.

```
//Saldo inicial del súper
wallet(math.round(math.random(1000))+500).
```

2. Owners

2.1. Colaboración/cooperación solicitud de cervezas y pinchos

En esta tarea tratamos la acción de que los Owners pudieran ir a por sus propias cervezas, sin necesidad de tener que pedírselo a myRobot, para esto utilizamos un número random que va ser el que, mediante una condición, decida si el Owner va a pedir la cerveza a myRobot o vaya él mismo.

```
+!drink(beer) : not has(myOwner,beer) & not thrown(garb1) & not asked(beer) <-
    ?id(Id);
    .println("Owner no tiene cerveza.");
    .random(N);
    if(N > 0.1){
        !get(beer);
    }else{
        .send(myRobot, askOne, turno(T), turno(T));
        .println(T);
        if(T == Id){
            .send(myRobot, achieve, changeTurn(T));
        }
        !pickFood;
    }
    !drink(beer).
```

Generamos un número random entre 0 y 1, con la función *.random(N)*, donde *N* es el número generado. Seguidamente, hacemos un if para decidir si el propio Owner va a por su cerveza o le pide a myRobot que vaya él. Si *N* es menor que 0.1, irá el Owner. Hay un 10% de posibilidades de que lo haga.
Cabe tener en cuenta que al tener dos Owners ha habido que implementar un sistema de turnos. De esta forma, myRobot sabrá qué petición debe atender, Owner1 u Owner2.

Al darse la situación, el Owner primero comprueba a quién está asignado el turno de myRobot, de esta forma: *.send(myRobot, askOne, turno(T), turno(T))*. Sii es su turno *if(T == Id)*, le cedemos el turno al otro Owner ya que no vamos a necesitar la atención de myRobot. Esto lo haremos enviándole a myRobot el goal de realizar el plan *changeTurn.* Finalmente, realizamos el plan *!pickFood,* que va a ser el encargado de realizar las acciones para que el Owner vaya a la nevera él mismo.

```
}else{
    .send(myRobot, askOne, turno(T), turno(T));
    if(T == Id){
        .send(myRobot, achieve, changeTurn(T));
    }
    !pickFood;
```

```
+!changeTurn(N)<-
    .abolish(turno(_));
    if(N == 1){+turno(2);}
    elif(N == 2){+turno(1);}.
```

<-myRobot.asl

Este plan, *!pickFood*, se realizará si no hay ninguna cerveza tirada, ya que como después veremos, el Owner también es capaz de recoger sus propias botellas y, por lo tanto podría dar problemas de concurrencia. El Owner va a realizar las mismas acciones que haría myRobot si tuviera que ir él a por la cerveza. Primero, el Owner irá hasta la nevera, *!go_at(myOwner,fridge)*, luego abre la nevera *open(fridge),* y seguidamente coge una cerveza y un pincho de la nevera, *get(beer), get(pincho*), estas funciones, que están en el fichero MyHouseModel.java sirven para restar una cerveza y un pincho, y así visualizarlo correctamente por pantalla. Luego, volvemos a restar la cerveza y el pincho , pero esta vez, en nuestro agente myFridge, *.send(myFridge, achieve*, *getOne(beer,Brand)), .send(myFridge, achieve, getOne(pincho,Pincho))*.
Hemos creado dos “neveras”, una en jason y otra en java. La de jason nos es mucho más cómoda a la hora de saber cuándo hay que comprar más alimentos para la nevera, y la de java nos permite visualizar en el grid el stock. Ambas neveras están totalmente coordinadas, ya que si introducimos o quitamos cervezas o pinchos en la nevera, realizamos las acciones seguidamente unas de otras a lo largo del código como vas a poder observar. Después, cierra la nevera una vez cogido lo necesario, *close(fridge)*, vuelve a la base *!go_at(myOwner,base)*, y una vez llegado a la base mediante las funciones *hand_in(beer)* y *hand_in(pincho)*, son las que le van a permitir comer el pincho y beber la cerveza.

```
+!pickFood: not thrown(garb1)<-
    .println("Voy a ir a la nevera yo mismo");
    !go_at(myOwner,fridge);
    .println("Voy a abrir la nevera");
    open(fridge);
    get(beer);
    get(pincho);
    .send(myFridge, achieve, getOne(beer,Brand));
    .send(myFridge, achieve, getOne(pincho,Pincho));
    .println("Voy a cerrar la nevera");
    close(fridge);
    !go_at(myOwner,base);
    hand_in(beer);
    hand_in(pincho).
```

Ahora, explicaremos más en detalle las funciones relacionadas con el fichero MyHouseModel.java y el MyHouseEnv.java que se utilizan en este plan, *get(beer), get(pincho*), *hand_in(beer)*, *hand_in(pincho).*
Para *get(beer)* y *get(pincho)*, enlazamos jason con java en el archivo MyHouseEnv.java donde mediante la función *executeAction(String ag, Structure action)*, esta función reconoce el agente, *ag*, que envía la acción y el nombre de la acción, *action*. Mediante ese nombre de agente y esa acción, los resuelve mediante condiciones y ejecuta una acción en MyHouseModel.java según ese *ag* y esa *action*. En este caso, nuestro *ag* sería nuestro Owner y *action* sería *get(beer)* y *get(pincho)*. Con esta información, esta función resuelve y ejecuta las funciones *getBeer()* y *getPincho()* de MyHouseModel.java.
La función *getBeer()* resta la variable *availableBeers* en uno y pone a true la variable *carryingBeer*. De esta manera, se sabe que el owner está llevando la cerveza consigo y lo mismo con *getPincho().* Resta la variable *availablePinchos* en uno y pone a true la variable *carryingPincho,* de esta manera se sabe que el owner está llevando el pincho consigo.

```
boolean getBeer() {
    availableBeers--;
    carryingBeer = true;

    if (view != null)
        view.update(lFridge.x,lFridge.y);
    return true;
}
```

```
boolean getPincho() {
    availablePinchos--;
    carryingPincho = true;

    if (view != null)
       view.update(lFridge.x,lFridge.y);
    return true;
}
```

Ahora, explicaremos las funciones *hand_in(beer)* y *hand_in(pincho), que* son las que permiten que el Owner pueda comenzar a beber cerveza, ya que, en ellas, ponemos los contadores de “sips” de cervezas y “eats” de pinchos a 10.

```
boolean handInBeer() {
    if (carryingBeer) {

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

```
boolean handInPincho() {
    if (carryingPincho) {

        eatCount = 10;
        carryingPincho = false;
        return true;
    } else {
        return false;
    }
}
```

Estos contadores son utilizados en el fichero MyHouseEnv.java para permitir enviar una creencia al Owner y comenzar a beber o comer.

```
public static final Literal hob = Literal.parseLiteral("has(myOwner,beer)");


 if (model.sipCount > 0) {
     addPercept("myRobot", hob);
     addPercept("myOwner", hob);
 }

public static final Literal hop = Literal.parseLiteral("has(myOwner,pincho)");

 if (model.eatCount > 0) {
     addPercept("myRobot", hop);
     addPercept("myOwner", hop);
 }
```

Aparte de eso, volvemos a poner a *false carryingBeer o carryingPincho*, ya que cuando se ejecuten estas acciones el Owner ya estará en su base dispuesto a beber o comer.

3. Myrobot

3.1. Servir cervezas y pinchos al owner

El Owner, cada vez que acaba de beber su cerveza va a pedir una nueva hasta llegar al límite establecido de 5 cervezas.

```
/* Initial beliefs and rules */
available(beer,fridge).
limit(beer,5).
```

Entonces, nos situamos en myOwner.asl. Se ejecuta el plan !drink(beer) siempre y cuando el owner no tenga cerveza, no haya lanzado una lata, y no haya pedido una nueva cerveza.
Tal y como comentábamos en el punto 2.1 como tanto myRobot como el Owner pueden ir a la nevera hemos generado un número random entre 0 y 1 para tomar esta decisión.
Si el número es mayor que 0.1, el Owner va pedir a myRobot que vaya él al fridge, es decir, hay un 90% de posibilidades de que myRobot vaya a por la cerveza. Si esta condición sucede, el Owner va a realizar el plan *!get(beer)* es decir, el owner va a pedir la cerveza al robot.

```
+!drink(beer) : not has(myOwner,beer) & not thrown(garb1) & not asked(beer) <-
    .println("Owner no tiene cerveza.");
    .random(N);
    if(N > 0.1){
        !get(beer);
    }else{
        .send(myRobot, askOne, turno(T), turno(T));
        .println(T);
        if(T == 1){
            .send(myRobot, achieve, changeTurn(T));
        }
        !pickFood;
    }
    !drink(beer).
```

En el plan *!get(beer)* de myOwner.asl que se ejecuta cuando todavía no se ha pedido una cerveza se manda a myRobot el goal de realizar el plan, *bring(Id,beer, Brand, Pincho)*, con la siguiente información: el *Id* del agente. 1, para myOwner y 2, para myOwner2; el producto (*beer)*, la marca de cerveza solicitada (*Brand)*, y el tipo de pincho y (*Pincho)*. En este plan, también le enviamos dinero al agente myRobotDel, que es el encargado de realizar la compra. De esta manera nos aseguramos de que siempre tenga dinero suficiente para realizar las compras cuando sean necesarias. Por último, se añade una creencia *asked(beer)*, para tener conocimiento de que se ha realizado el pedido de cerveza.

```
+!get(beer) : not asked(beer) <-
    ?wallet(M);
    ?beer_brand(Brand);
    ?pinchoType(Pincho);
    .send(myRobot, achieve, bring(1,beer, Brand, Pincho));
    .send(myRobotDel, achieve, send_money(M));
    .println("Owner ha pedido una cerveza al robot acompañado de un pincho.");
    +asked(beer).
```

El plan *bring(Id,beer, Brand, Pincho)* de myRobot.asl simplemente añade a su base de creencias la creencia *+asked(Ag, Id, P, Brand, Pincho)* con todos los datos necesarios del pedido. Ahora ya podrá ejecutarse el plan *!bringBeer.* Este se llevará a cabo siempre que se le haya hecho un pedido, que el Owner que lo ha realizado tiene el turno de atención y que no se haya llegado al límite de 5 cervezas. En el plan *!bringBeer* myRobot irá a la nevera *!go_at(myRobot,fridge)*, luego ejecutará el plan *!take(Ag,fridge,beer, Brand, Pincho)* y se lo llevará al Owner correspondiente. Además, se llama al plan *!hasBeer(myOwner)*. Por último cambiará el turno de atención al otro Owner (ver el if).

```
+!bring(Id, P, Brand, Pincho)[source(Ag)] <-
    +asked(Ag, Id, P, Brand, Pincho).
```

```
+!bringBeer : asked(Ag,Id,beer, Brand, Pincho) & turno(Id) & not healthMsg(_)<-
    .println("Owner me ha pedido una cerveza acompañado de un pincho.");
    !go_at(myRobot,fridge);
    !take(Ag,fridge,beer, Brand, Pincho);
    !go_at(myRobot,Ag);
    !hasBeer(Ag);
    .println("Ya he servido la cerveza y el pincho y elimino la petición.");
    .abolish(asked(Ag,Beer, Brand, Pincho));
    .abolish(turno(Id));
    if(Id == 1){
        +turno(2);
    }elif(Id == 2){
        +turno(1);
    }
    !bringBeer.
```

El plan *!hasBeer(myOwner)* simplemente le da la cerveza/pincho al Owner *hand_in(Producto)* y añade la creencia de que se ha consumido una cerveza más con su fecha y hora de consumo.

```
+!hasBeer(myOwner) : not too_much(beer) <-
    hand_in(beer);
    hand_in(pincho);
    .println("He preguntado si Owner ha cogido la cerveza y el pincho.");
    .println("Se que Owner tiene la cerveza.");
    .date(YY,MM,DD); .time(HH,NN,SS);
    +consumed(YY,MM,DD,HH,NN,SS,beer).
```

El plan *!take(Ag,fridge,beer, Brand, Pincho),* se ejecuta siempre que el Owner pueda seguir bebiendo cervezas. En él se comprueba el stock de la nevera. Si el número de cervezas o pinchos es menor a 6 se solicitará a myRobotDel que reponga el stock haciendo una compra al supermercado. En la última línea se llama al plan *!check,* explicado a continuación

```
+!take(Ag,fridge, P, Brand, Pincho) : not too_much(beer)<-
    .println("El robot está cogiendo una cerveza y un pincho para acompañar.");
    .send(myFridge, askOne, stock(P, Brand, S), stock(P, Brand, S));
    .send(myFridge, askOne, stock(pincho, Pincho, Sp), stock(pincho, Pincho, Sp));
    if(S < 6 | Sp < 6){
        .send(myRobotDel, achieve, buy(P, Brand));
        .send(myRobotDel, achieve, buy(pincho, Pincho));
    }
    !check(Ag,fridge, P, Brand,Pincho).
```

En este plan, *!checkcheck(myOwner,fridge, P,Brand,Pincho)*, que sería uno para cada Owner, el agente MyRobot abre la nevera *open(fridge)* y *coge la cerveza* y el pincho*, get(beer), get(pincho)*.

```
+!check(myOwner,fridge, P,Brand,Pincho)<-
    .println("El robot está en el frigorífico y coge una cerveza.");
    .wait(100);
    open(fridge);
    .println("El robot abre la nevera.");
    .wait(1000);
    get(beer);
    get(pincho);
    .send(myFridge, achieve, getOne(beer,Brand));
    .send(myFridge, achieve, getOne(pincho,Pincho));
    .println("El robot coge una cerveza y un pincho.");
    close(fridge);
    .println("El robot cierra la nevera.").
```

Estas tres funciones (*open(fridge),get(beer), get(pincho)*) se gestionan a través myHouseEnv.java que a su vez llamará a sus respectivas funciones *openFridge(), getBeer(), getPincho()* del myHouseModel.java

```
public static final Literal of  = Literal.parseLiteral("open(fridge)");
public static final Literal gb1  = Literal.parseLiteral("get(beer)");
public static final Literal gp1  = Literal.parseLiteral("get(pincho)");
```

```
}else if(action.equals(of) & ag.equals("myOwner")){
    result = model.openFridge();
}else if(action.equals(gb1) & ag.equals("myOwner")){
    result = model.getBeer();
}else if(action.equals(gp1) & ag.equals("myOwner")){
    result = model.getPincho();
```

En *openFridge(),* si la nevera está cerrada (!fridgeOpen) se abre, en *getBeer()* se actualiza el número de cervezas disponibles restándole una a *avaliableBeers* y en *getPincho()* lo mismo con el número de pinchos *avaliablePinchos*. Además, se actualiza la vista de la nevera para que se vea su stock actualizado.

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

```
boolean getBeer() {
        availableBeers--;
        carryingBeer = true;

        if (view != null)
            view.update(lFridge.x,lFridge.y);
        return true;

}
```

```
boolean getPincho() {

        availablePinchos--;
            carryingPincho = true;

        if (view != null)
            view.update(lFridge.x,lFridge.y);
        return true;


}
```

Volviendo al plan *!check(myOwner,fridge, P,Brand,Pincho)* de myRobot, nos quedamos en que el robot envía a myFridge el goal de realizar los planes *getOne(beer,Brand))*, *getOne(pincho,Pincho))*. Por último cierra la nevera.

```
+!check(myOwner,fridge, P,Brand,Pincho)<-
    .println("El robot está en el frigorífico y coge una cerveza.");
    .wait(100);
    open(fridge);
    .println("El robot abre la nevera.");
    .wait(1000);
    get(beer);
    get(pincho);
    .send(myFridge, achieve, getOne(beer,Brand));
    .send(myFridge, achieve, getOne(pincho,Pincho));
    .println("El robot coge una cerveza y un pincho.");
    close(fridge);
    .println("El robot cierra la nevera.").
```

En el plan *getOne(Product,Brand))* de myFridge.asl, la nevera actualiza el stock restando una cerveza/pincho. Este plan sólo se ejecuta si la nevera tiene suficiente cantidad para satisfacer el pedido.

```
+!getOne(P, Brand): stock(P,Brand,Qtd) & Qtd > 0<-
    -stock(P, Brand, S);
    +stock(P, Brand, S-1).
```

Por último *close(fridge)* se realiza con la misma lógica que *open(fridge).*

3.2. Recoger los platos sucios del owner y llevarlos al lavavajillas

Para esta tarea, usamos un agente específico, myRobotDWash. En su fichero .asl, dentro del plan *!drink(beer)* que se ejecuta cuando el owner tiene una cerveza, un pincho, todavía no ha lanzado la botella ni ha pedido una nueva cerveza, vamos a ejecutar la función de java *throwPincho(dish).*

```
+!drink(beer) : has(myOwner,beer) & not thrown(garb1)& has(myOwner,pincho) & not asked(beer) <-
    sip(beer);
    .println("Owner está bebiendo cerveza.");
    eat(pincho);
    .println("Owner está comiendo el pincho.");
    throwBottle(garb);
    throwPincho(dish);
    !drink(beer).
```

En el fichero MyHouseEn.java se comprueba que esta acción es solicitada por el Owner y se llama a la función *throw_dish()* del fichero MyHouseModel.java

```
public static final Literal tp = Literal.parseLiteral("throwPincho(dish)");
```

```
}else if(action.equals(tp) & ag.equals(anObject:"myOwner")){
    result = model.throw_dish();
```

En dicha función del myHouseModel.java se comprueba que se hayan dado todos los bocados al pincho *eatCount=0* y se pone la variable *thrownDish* a *true*.

```
boolean throw_dish(){
    if(eatCount == 0){
        thrownDish = true;
        return true;
    }else{
        return false;
    }
}
```

A su vez, en myHouseEnv.java se está comprobando que efectivamente se haya lanzado el plato, *thrownDish = true* y se añade al robot myRobotDWash, la creencia *td*, *thrown(dish)*.

```
public static final Literal td = Literal.parseLiteral("thrown(dish)");

if(model.thrownDish){
    addPercept("myRobotDWash", td);
}
```

Gracias a esta creencia, podemos hacer que nuestro agente de recogida de platos, ejecute el plan *!CleanDish.* En este plan, va a junto del Owner *!go_at(myRobotDWash, myOwner)* y recoge el plato usando la función *pickUpDish(dish)*. Esto solo pasará en caso de que tengamos la creencia *thrown(dish)* y además el lavavajillas no haya acabado con el lavado, ya que, en este caso el lavavajillas estaría lleno y el robot esperaría a que se vacíe y se puedan insertar más platos.

```
+!cleanDish: thrown(dish) & not finishDWash<-
    .println("Voy a recoger el plato vacío");
    .wait(100);
    !go_at(myRobotDWash,myOwner);
    pickUpDish(dish);
    !go_at(myRobotDWash,dwash);
    .send(myDishWasher,achieve,addDish);
    .abolish(thrown(dish));
    .wait(100);
    !go_at(myRobotDWash,base);
    !cleanDish.
```

Desde el fichero MyHouseEnv.java se llama a *pickUpDish()* del MyHouseModel.java.

```
public static final Literal pd = Literal.parseLiteral("pickUpDish(dish)");

}else if(action.equals(pd) & ag.equals(anObject:"myRobotDWash")){
    result = model.pickUpDish();
```

En MyHouseModel.java usamos esta función para devolver la variable *thrownDish* a *false*.

```
boolean pickUpDish(){
    if(thrownDish){
        thrownDish = !thrownDish;
        return true;
    }else{
        return false;
    }
}
```

Además, siguiendo en el plan *!CleanDish* del fichero myRobotDWash.asl el agente se mueve hacia el lavavajillas *!go_at(myRobotDwash, dwash)* y se le envía el goal (al lavavajillas) de realizar el plan *addDish* con *.send(myDishWasher,achieve,addDish)*. A continuación, nos deshacemos de la creencia *thrown(dish)* y llevamos al robot a su base.

```
+!cleanDish: thrown(dish) & not finishDWash<-
    .println("Voy a recoger el plato vacío");
    .wait(100);
    !go_at(myRobotDWash,myOwner);
    pickUpDish(dish);
    !go_at(myRobotDWash,dwash);
    .send(myDishWasher,achieve,addDish);
    .abolish(thrown(dish));
    .wait(100);
    !go_at(myRobotDWash,base);
    !cleanDish.
```

Este sería el plan !*addDish* en myDishWasher.asl, en él se actualiza el número de platos añadiendo uno más.

```
+!addDish<-
     ?dishes(N);
     -dishes(N);
     addDish(dish);
     +dishes(N+1).
```

3.3. Llevar los platos limpios a la alacena.

En myDishWasher.asl que es el lavavajillas, tenemos el plan *!washDishes* que se ejecuta cuando el número de platos es igual a tres. En él, se actualiza el nº de platos a cero, así el robot que se encarga de recoger los platos ya puede traer nuevos platos sucios. Después de un *wait* que simula un lavado, se envía al robot la creencia de que ya ha terminado, *.send(myRobotDWash, tell, finishDWash)*

```
+!washDishes: dishes(S) & S == 3<-
    -dishes(S);
    +dishes(0);
    .println("El lavavajillas está lleno. Vamos a ponerla a funcionar.");
    .wait(100);
    .println("Platos lavados!!");
    .send(myRobotDWash, tell, finishDWash);
    !washDishes.
```

En el caso del robot se ejecuta el plan *!cleanDish* cuando tiene dicha creencia (*finishDWash)*. Primero eliminamos la creencia, seguidamente el robot se desplaza al lavavajillas y después a la alacena *!go_at(myRobotDWash,dwash); !go_at(myRobotDWash,cboard)*. LLamamos a la función de java *emptyDW(dishes)* y *addCBoard(dishes*) que se encargarán de actualizar la vista del número de platos que hay en el lavavajillas a cero y la de la alacena a tres, que es el número de platos limpios que se sacan.

```
+!cleanDish: finishDWash<-
    .abolish(finishDWash);
    .println("Voy a recoger los platos lavados y llevarlos a la alacena");
    !go_at(myRobotDWash,dwash);
    !go_at(myRobotDWash,cboard);
    emptyDW(dishes);
    .println("Voy a dejarlos en alacena.");
    addCBoard(dishes);
    !go_at(myRobotDWash,base);
    !cleanDish.
```

```
boolean addCB(){
    dishCountCB += 3;
    if (view != null)
        view.update(lCBoard.x,lCBoard.y);
    return true;
}
```

```
boolean emptyDW(){
    dishCount = 0;
    if (view != null)
        view.update(lDWash.x,lDWash.y);
    return true;
}
```

4. Cleaner

4.1. Recoger las latas del grid, colaboración entre Owner y myCleaner

Las latas serán recogidas por el robot myCleaner, éste actuará cuando alguno de los dos owners lance una lata. Sin embargo, en algunos casos también puede ser el owner quien recoja sus propias latas.

En primer lugar veremos cuando el owner lanza una lata:
Dentro del fichero myOwner.asl tenemos el plan *!drink(beer)* que se ejecuta cuando el owner tiene una cerveza, un pincho, todavía no ha lanzado la botella ni ha pedido una nueva cerveza. En dicho plan se ejecuta la funcion en java *throwBottle(garb)*

```
+!drink(beer) : has(myOwner,beer) & not thrown(garb1)& has(myOwner,pincho) & not asked(beer) <-
    sip(beer);
    .println("Owner está bebiendo cerveza.");
    eat(pincho);
    .println("Owner está comiendo el pincho.");
    throwBottle(garb);
    throwPincho(dish);
    !drink(beer).
```

En el fichero MyHouseEnv.java se comprueba que esta acción es solicitada por el Owner y se llama a la función *throw_bottle()* del fichero MyHouseModel.java

```
public static final Literal tb = Literal.parseLiteral("throwBottle(garb)");
```

```
}else if(action.equals(tb) & ag.equals("myOwner")){
    result = model.throw_bottle();
```

En dicha función del Model.java se comprueba que se hayan dado todos los sorbos a la cerveza, *sipCount==0*. En cuyo caso, se genera una nueva ubicación para la lata, *lB,* comprobando siempre que esta, no coincida con ningún obstáculo (siendo *lObstacles* una lista con las posiciones de todos los obstáculos) tal y como podemos ver en el do-while, a demás para que las latas tampoco se coloquen encima de los Owners u otros elementos de la casa, la posición generada estará dejando un marco de una casilla. Añadimos la posición escogida al array de botellas *lBottle* con *lBottle.add(lB).*

Ahora veremos quién va a recoger la lata del suelo, o bien el robot myCleaner o bien, el Owner que la lanzó.
Para esta funcionalidad hay que fijarse en la variable *randomNum* que genera un número aleatorio entre 0 y 1 y además en la variable *thrownBottle* que se pone a *true*.

```
boolean throw_bottle(){
    if(sipCount == 0){
        randomNum = new Random().nextDouble();
        Location lB = new Location(0,0);
        do{
            lB = new Location(new Random().nextInt(8)+1,new Random().nextInt(8)+1);
        }while(lObstacles.contains(lB));
        add(BOTTLE,lB);
        lBottle.add(lB);
        thrownBottle = true;
        if(lBottle.size() > 0){
            for(Location i : lBottle){
                view.update(i.x,i.y);
            }
        }
        return true;
    }else{
        return true;
    }
}
```

Ambas variables: *thrownBottle* y *RandomNum* se usarán en myHouseEnv.java para decidir quién irá a por la lata. Por un lado, *thrownBottle* se utiliza para comprobar que efectivamente se haya lanzado la lata, *thrownBottle* = *true* y por otro lado *RandomNum* se usa para tomar la decisión. Como vemos, hay un 10% de posibilidades de que vaya el Owner a recogerla.
Una vez tomada la decisión mediante el número aleatorio previamente generado, se añade la creencia de ir a recoger la lata (*tg1*) es decir, *thrown(garb1)* al elegido.

```
public static final Literal tg1 = Literal.parseLiteral("thrown(garb1)");
```

```
if(model.thrownBottle){

    if(model.randomNum > 0.1){
        addPercept("myCleaner", tg1);
    }else{
        addPercept("myOwner", tg1);
    }

}
```

Gracias a esta creencia, podemos hacer que nuestro agente vaya a buscar la lata *!go_at(Ag,garb1)* y la recoja usando la función *pickUpBottle(garb1)*.

En caso de que el elegido sea myCleaner:

```
+!cleanBottle : thrown(garb1)<-
    .println("pick bottle 1");
    !go_at(myCleaner,garb1);
    pickUpBottle(garb1);
    !go_at(myCleaner,paper);
    thrashBottle(garb);
    .wait(100);
    .println("Botella echada en la basura");
    .abolish(thrown(garb1));
    !go_at(myCleaner,base);
    !cleanBottle.
```

En caso de que el elegido sea myOwner:

```
+!cleanBottle : thrown(garb1)<-
    .println("Voy yo mismo a recoger mi botella");
    !go_at(myOwner,garb1);
    pickUpBottle(garb1);
    !go_at(myOwner,paper);
    thrashBottle(garb);
    .wait(100);
    .println("Botella echada en la basura");
    .abolish(thrown(garb1));
    !go_at(myOwner,base);
    !cleanBottle.
+!cleanBottle<-!cleanBottle.
```

La función *pickUpBottle()* comprueba que efectivamente se ha lanzado una lata y hay que recogerla, *thrownBottle=true.* Si esto es cierto, se devuelve la variable thrownBottle a false para indicar que ya no hay que recoger la lata. Seguidamente eliminamos el objeto lata *remove(BOTTLE,lBottle.get(0))* para que así se elimine de la vista, después, eliminamos su posición del array de posiciones de las latas lBottle a través de *lBottle.remove(0).*

```
boolean pickUpBottle(){
    if(thrownBottle){
        thrownBottle = !thrownBottle;
        remove(BOTTLE,lBottle.get(0));
        lBottle.remove(0);
        atBottle = false;
        return true;
    }
    return false;
}
```

4.2. Llevar las latas a la papelera

Tal y como hablábamos anteriormente. Tanto Owner como myCleaner pueden recoger latas del grid. Por esto, ambos podrán llevarlas a la papelera y lo harán de la misma forma. Por ello, para explicar este apartado me centraré en cómo lo hace myCleaner. Una vez ha recogido la lata a través de la función *pickUpBottle(garb1),* se mueve hacia la la papelera *!go_at(myCleaner,paper)* y ejecuta la función en java *thrashBottle(garb).*

```
+!cleanBottle : thrown(garb1)<-
    .println("pick bottle 1");
    !go_at(myCleaner,garb1);
    pickUpBottle(garb1);
    !go_at(myCleaner,paper);
    thrashBottle(garb);
    .wait(100);
    .println("Botella echada en la basura");
    .abolish(thrown(garb1));
    !go_at(myCleaner,base);
    !cleanBottle.
```

Dicha función del fichero MyHouseEnv.java, llama a la función *trashBottle(garb)* del archivo MyHouseModel.java.

```
public static final Literal thrashb = Literal.parseLiteral("thrashBottle(garb)");
```

```
}else if(action.equals(thrashb) & ag.equals("myCleaner")){
    result = model.trashBottle();
```

En MyHouseModel.java, *trashBottle()* incrementa en uno el número de latas que hay en la papelera (*bottleCount).* Esto nos servirá más adelante para identificar cuando la papelera está llena. Por último, se actualiza la view para que se refleje el número de latas que hay en la papelera.

```
boolean trashBottle(){
    bottleCount++;
    if (view != null)
            view.update(lPaper.x,lPaper.y);
    return true;
}
```

5. Burner

5.1. Vaciar el contenido de la papelera cuando se llene

Hay que tener en cuenta que la papelera será vaciada por myRobotPaper cuando se llene, es decir, cuando tenga 5 latas dentro.
En el punto 4.2 explicábamos que una vez que el robot/owner ha recogido del suelo la lata y va a la papelera ejecuta la función *trashBottle(garb)* del archivo MyHouseEnv.java.

```
+!cleanBottle : thrown(garb1)<-
    .println("pick bottle 1");
    !go_at(myCleaner,garb1);
    pickUpBottle(garb1);
    !go_at(myCleaner,paper);
    thrashBottle(garb);
    .wait(100);
    .println("Botella echada en la basura");
    .abolish(thrown(garb1));
    !go_at(myCleaner,base);
    !cleanBottle.
```

Dicha función del fichero MyHouseEnv.java, llama a la función *trashBottle()* del archivo MyHouseModel.java.

```
public static final Literal thrashb = Literal.parseLiteral("thrashBottle(garb)");
```

```
}else if(action.equals(thrashb) & ag.equals("myCleaner")){
    result = model.trashBottle();
```

trashBottle(), realiza el conteo de botellas que se arrojan a la basura a través de la variable *bottleCount*. Y actualiza la vista de la papelera.

```
boolean trashBottle(){
    bottleCount++;
    if (view != null)
        view.update(lPaper.x,lPaper.y);
    return true;
}
```

Mientras tanto, en el archivo MyHouseEnv.java se estará comprobando si el número de botellas de la papelera *bottleCount* llega a 5. En cuyo caso, se añade la percepción *paperFull* (*pF*) al agente myRobotPaper.

```
public static final Literal pF = Literal.parseLiteral("paperFull");
```

```
if(model.bottleCount == 5){
    addPercept("myRobotPaper", pF);
}
```

Ahora, nos situamos en el fichero myRobotPaper.asl. Lo primero que podemos ver es que el agente tiene un goal (objetivo) inicial, *!emptyPaper()*. Que se ejecutará siempre y cuando se cumpla la condición de que la papelera está llena (*paperFull*) que hemos visto antes.

```
/* Initial goals */
!emptyPaper.

/* Plans */
+!emptyPaper : paperFull <-
    .print("La basura está llena, voy a quemar la basura");
    burning(garb);
    .wait(3000);
    emptyPaper;
    burning(garb);
    .print("Ya he quemado la basura");
    !emptyPaper.
+!emptyPaper<-!emptyPaper.
```

Como vemos, en el plan *!emptyPaper,* se llama a la función de java *burning(garb),* hace un *wait(3000)* y vuelve a llamar a *burning(garb)*. La función de esto será crear la animación rojo-verde cuando el robot myRobotPaper esté vaciando la papelera o no. A través de *burning(garb)* ese llamara a la función *burnGarb()* del MyHouseModel.java

```
public static final Literal burning = Literal.parseLiteral("burning(garb)");

}else if(action.equals(burning) & ag.equals("myRobotPaper")){
    result = model.burnGarb();
```

La función *burnGarb(), pasará la variable burningGarb a false* si estaba a *true* y viceversa.

```
boolean burnGarb(){
    burningGarb = !burningGarb;
    return true;
}
```

Continuando con lo anterior, volvemos al fichero myRobotPaper.asl. Después de realizar la animación, se llama a *emptyPaper.*

```
+!emptyPaper : paperFull <-
    .print("La basura está llena, voy a quemar la basura");
    burning(garb);
    .wait(3000);
    emptyPaper;
    burning(garb);
    .print("Ya he quemado la basura");
    !emptyPaper.
```

De esta forma se llamará a la función *emptyPaper()* del myHouseModel.java

```
public static final Literal eP = Literal.parseLiteral("emptyPaper");
```

```
}else if(action.equals(eP) & ag.equals("myRobotPaper")){
    result = model.emptyPaper();
```

emptyPaper() devolverá el número de latas dentro de la papelera (*bottleCount)* a 0 y actualizará la vista de la misma.

```
boolean emptyPaper(){
    bottleCount = 0;
    if (view != null)
        view.update(lPaper.x,lPaper.y);
    return true;
}
```

6. Obstáculos

6.1. Generación aleatoria de obstáculos

Para este apartado en el fichero MyHouseModel.java hemos generado entre 7 y 9 posiciones aleatorias para los obstáculos. Estas se generan dejando un marco de dos posiciones para que no coincidan con los elementos como nevera, alacena, owners, etc. y tampoco les estorben colocándose justo delante . En el bucle *while* comprobamos que la posición creada no coincide ni con un obstáculo ya creado ni con la posición inicial de myRobot. En caso de que sí coincida con alguno, se genera una nueva posición y así sucesivamente, una vez se haya generado una posición válida se añade al array.

```
ArrayList<Location> lObstacles = new ArrayList<>();


for (int index = 0; index < 9; index++) {
    Location lObstacle = new Location(new Random().nextInt(6)+2,new Random().nextInt(6)+2);

    while(lObstacle == lRobot  || lObstacles.contains(lObstacle)){
        lObstacle = new Location(new Random().nextInt(6)+2,new Random().nextInt(6)+2);
    }

    lObstacles.add(lObstacle);
}
```

Para visualizarlo en el grid, recorremos el array de obstáculos y usamos la función *addWall()* para representar cada uno. De esta forma:

```
for (int i = 0; i < lObstacles.size(); i++) {
    int x = lObstacles.get(i).x;
    int y = lObstacles.get(i).y;
    addWall(x,y,x,y);
}
```

6.2. Los agentes deben esquiva Obstáculos generados, Otros agentes y Owners Nevera/alacena/lavavajillas/papelera

Para implementar esto hemos creado dos funciones en MyHouseModel.java, la primera denominada *IsOcupiedbyRobot(Location l)*. En ella creamos un ArrayList en el que iremos metiendo todas las posiciones de los robots. Devuelve *true* si en la posición que se pasa como parámetro hay un robot y *false* en caso contrario.

```
boolean isOccupiedByRobot(Location lo){
    List<Location> list = new ArrayList<>();
    for (int i = 0; i < numRobots; i++) {
        list.add(getAgPos(i));
    }

    if(list.contains(lo)){
        return true;
    }else{
        return false;
    }
}
```

y *getAdjacentLocations(Location l)* que devuelve una lista con las posiciones adyacentes a la posición pasada como parámetro (chequeando siempre que no se salga del grid). Para que no se guarden siempre en el mismo orden, hacemos un *.shuffle*.

```java
public List<Location> getAdjacentLocations(Location loc) {
    List<Location> adjacents = new ArrayList<>();

    // Check up
    if (loc.y > 0) {
        adjacents.add(new Location(loc.x, loc.y - 1));
    }

    // Check right
    if (loc.x < GSize - 1) {
        adjacents.add(new Location(loc.x + 1, loc.y));
    }

    // Check down
    if (loc.y < GSize - 1) {
        adjacents.add(new Location(loc.x, loc.y + 1));
    }

    // Check left
    if (loc.x > 0) {
        adjacents.add(new Location(loc.x - 1, loc.y));
    }

    Collections.shuffle(adjacents);

    return adjacents;
}
```

Por otro lado, añadimos al *move_towards* de java el siguiente código.

```java
if (lObstacles.contains(r1) || isOccupiedByRobot(r1) || r1.equals(lDWash)
|| r1.equals(lCBoard) || r1.equals(lPaper) || r1.equals(lFridge)) {

    List<Location> adjacentPositions = getAdjacentLocations(getAgPos(ag));
    for (Location adj : adjacentPositions) {
        if (!lObstacles.contains(adj) && !isOccupiedByRobot(adj) && !adj.equals(lDWash)
        && !adj.equals(lCBoard) && !adj.equals(lPaper) && !adj.equals(lFridge) ) {
            r1 = adj;
            setAgPos(ag, r1);
            break;
        }
    }
} else {
    setAgPos(ag, r1);
}
```

r1: es la siguiente posición a la que el agente se va mover.
lObstacles: Lista con las posiciones de todos los obstáculos.
En caso de que *r1* sea una posición ya ocupada (por otro obstáculo,un robot, o algún elemento de la casa), creamos una lista con las posiciones adyacentes a la posición actual del agente e iteramos sobre ella para elegir aquella posición que nos sirva, es decir, en la que no haya ni un obstáculo ni un robot, ni otro elemento. Por último, movemos el agente. En caso de que en un primer momento *r1* sea una posición válida movemos el agente directamente.

7. Lógica de movimiento

7.1. Posicionado en casillas adyacentes y movimiento no diagonal.

Para realizar el movimiento diagonal, buscamos evitar que el índice X e Y aumenten simultáneamente. Por ello, usamos un *numeroRandom* entre 0 y 1 y, cuando sea mayor que 0.5 trabajamos sobre el eje X y viceversa.
Una vez elegido el eje, se compara la posición actual (*r1*) con la de destino final (*dest)*. Si el destino (*dest*) es mayor a la posición actual (*r1*), se incrementa el eje elegido en uno.
Por ejemplo, si estamos trabajando sobre el eje X, nos encontramos en la posición (3,2) y nuestro destino es la posición (4,2); incrementaremos X en uno llegando así a destino.

```
Random random = new Random();
double numeroRandom = random.nextDouble();

if (numeroRandom > 0.5) {
      // Moverse en el eje x
      if (dest.x > r1.x) {
        r1.x++;
      } else if (dest.x < r1.x) {
        r1.x--;
      }
} else {
      // Moverse en el eje y
      if (dest.y > r1.y) {
        r1.y++;
      } else if (dest.y < r1.y) {
        r1.y--;
      }
}
```

Para que el robot se coloque en posiciones adyacentes, simplemente creamos una variable *closeToX* con la posición en la que vamos a querer que se coloque el agente.

```
Location closeTolDelivery  = new Location(1,GSize-2);
Location closeTolFridge = new Location(0,1);
Location closeToDWash = new Location(GSize/2, 1);
Location closeTolOwnerDW = new Location(GSize-1, GSize-2);
Location closeTolOwner2DW = new Location(GSize-3, GSize-2);

Location closeToCBoard = new Location(lCBoard.x, 1);
Location auxFridge  = new Location(closeTolFridge.x,closeTolFridge.y+1);
Location closeTolOwner2 = new Location(GSize-4,GSize-1);
Location closeTolOwner  = new Location(GSize-2,GSize-1);
Location closeTolPaper = new Location(GSize-1,1);
```

Ahora, vamos a hacer que la variable *atX*, que usamos para lanzar una creencia al agente oportuno indicando que ha llegado a su destino, sea igual a las posiciones *closeToX* que hemos creado.

```
if(ag == 0){
    atOwner = r1.equals(closeTolOwner);
    atOwner2 = r1.equals(closeTolOwner2);
    atFridge = r1.equals(closeTolFridge);
    atDelivery = r1.equals(lDelivery);
    atBase = r1.equals(lRobot);
    atPaper = r1.equals(closeTolPaper);
}else if(ag == 1){
    atFridgeD = r1.equals(closeTolFridge);
    atDeliveryD = r1.equals(lDelivery);
    atBaseD = r1.equals(lDelRobot);
}else if(ag == 2){
    atDWash= r1.equals(closeToDWash);
    atCBoard = r1.equals(closeToCBoard);
    atOwnerDW = r1.equals(closeTolOwnerDW);
    atOwner2DW = r1.equals(closeTolOwner2DW);
    atBaseDw = r1.equals(lDWashRobot);
}else if(ag == 4){
    atBaseO = r1.equals(lOwner);
    atPaperO = r1.equals(closeTolPaper);
    atFridgeO = r1.equals(closeTolFridge);
}else if(ag == 5){
    atBaseCl = r1.equals(lCleaner);
    atPaperCl = r1.equals(closeTolPaper);
}else if(ag == 6){
    atBaseO2 = r1.equals(lOwner2);
    atPaperO2 = r1.equals(closeTolPaper);
    atFridgeO2 = r1.equals(closeTolFridge);
}
```