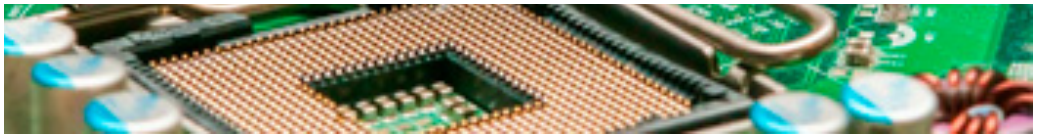
Tema 4: Agentes Inteligentes (2)



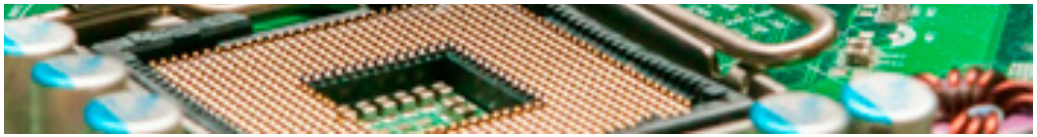
Agentes inteligentes

Modelos y arquitecturas de agentes inteligentes

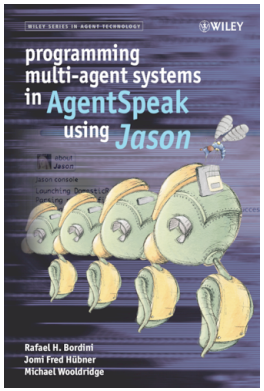
JASON: ejemplo de programa de agente

Representación del conocimiento. Ontologías

Comunicación y Sistemas multiagente



Bibliografía para Jason

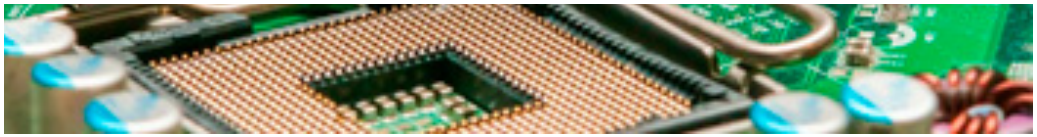


Programming Multi-Agent Systems in AgentSpeak using Jason

Rafael H. Bordini, Jomi Fred Hübner, Michael Wooldridge

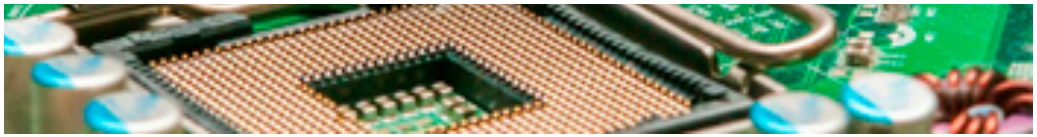
ISBN: 978-0-470-02900-8

http://jason.sourceforge.net/wp/



Jason

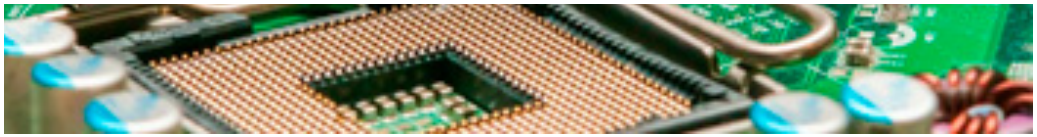
- El lenguaje que interpreta es una extensión de AgentSpeak (arquitectura BDI).
- Posibilidad de usar distintas infraestructuras de agentes (Jade, SACI,Magentix2, ...)
- Numerosas características configurables por el usuario.
- Disponible en código abierto.
- Vamos a ver su **Lenguaje**



Elementos del Lenguaje

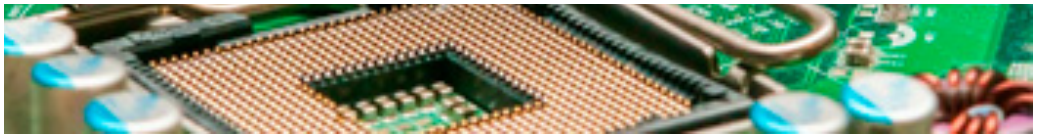
Los elementos fundamentales del Lenguaje son:

- Creencias (Beliefs)
- Objetivos (Goals)
- Planes (Plans, Intentions)



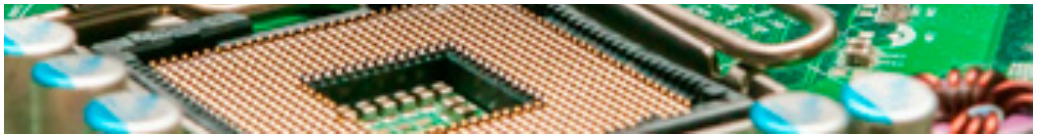
Beliefs

- Cada agente tiene una base de creencias
- Beliefs: Colección de literales representados como predicados.
    - tall(jhon).
    - likes(jhon, music)
- Anotaciones: detalles asociados a una creencia.
    - busy(jhon)[expires(autum)]
- Las anotaciones aportan '*elegancia*' al lenguaje y facilitan el manejo de la base de creencias.



Beliefs

- Existen anotaciones que tienen un signicado especial para el interprete. En particular la anotacion *source*
- Hay tres tipos de fuentes de informacion para los agentes
  - Informacion perceptual: aquella que percibe del entorno
    -source(percept)
  - Comunicacion: aquella que proviene de otro agente del sistema - source(id agente)
  - Notas mentales: creencias que provienen del propio agente - source(self)
- Ejemplos
  - Perceptual (colour(box1,blue)[source(percept)])
  - Comunicación (colour(box1,blue)[source(bob)])
  - Notas mentales (colour(box1,blue)[source(Self)])



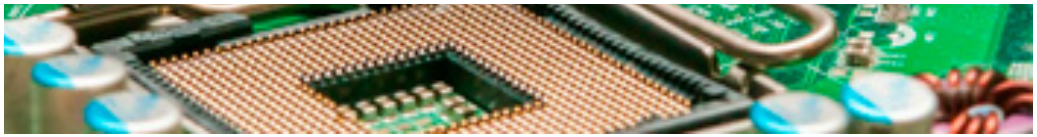
Beliefs

- StrongNegation:se denota con "~". Expresa que el agente cree explícitamente que algo es falso.

  ~colour(box1, white)[source(john)]

- Reglas:Permiten inferir nueva información a partir de conocimiento que se tiene:

```
likely_colour(C,B)
        :-colour(C,B)[source(S)] &
        (S == self | S == percept).
likely_colour(C,B)
        :-colour(C,B)[degOfCert(D1)] &
        not (colour(_,B)[degOfCert(D2)] & D2 > D1) &
        not~colour(C,B).
```

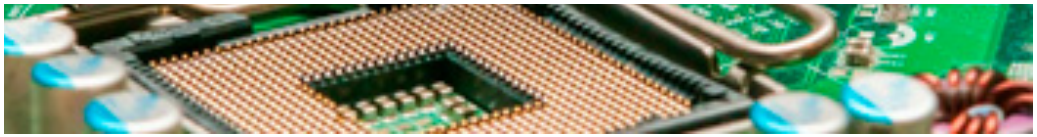



Goals

- Existen dos tipos de objetivos en Jason:
  - Achievement goals(operador !): Expresan un estado del mundo que el agente desea conseguir.

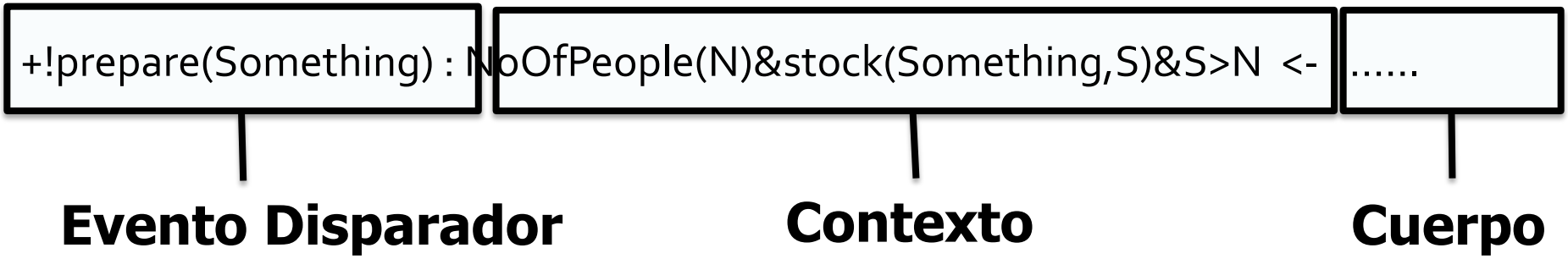
    !own(house)
  - Test goals(operador ?): Usados normalmente para recuperar información de la base de creencias.

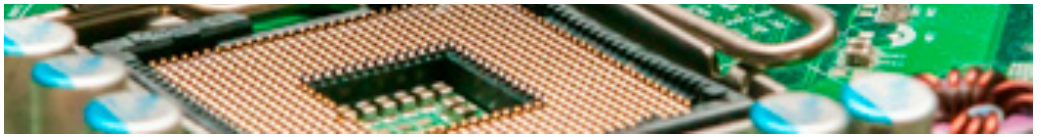
    ?bank_balance(BB)



Plans

- Un plan tiene tres partes
  - Triggering event: Cambios en las creencias o objetivos del agente
  - Context: Determinan si un plan es aplicable
  - Body: Sucesión de acciones que comporta el plan





Plans

+!prepare(Something) : NoOfPeople(N)&stock(Something,S)&S>N <- .....

Evento Disparador

Cambios en las creencias u objetivos

Contexto

Literales que deben ser una consecuencia lógica de la base de creencias para que el plan se instancie

Cuerpo

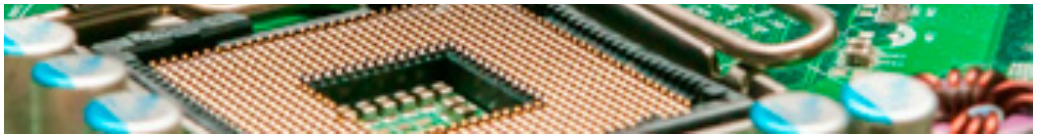
Secuencia de acciones. Pueden existir nuevos subobjetivos

+ L Belief addition
- L Belief deletion
+! L Achievement-goal addition
-! L Achievement-goal deletion
+? L Test-goal addition
-! L Test-goal addition

L The agent believes L is true
~ L The agent believes L is false
not L The agent does not believes L is true
not ~ L The agent does not believes L is false

Plan relevante

Plan aplicable



Plans - Cuerpo

Cuerpo del plan: Secuencia de instrucciones separadas por ";". Estas instrucciones puedes ser:

- Actions: Acciones externas que se denotan por un predicado. Proporcionan un "feedback".

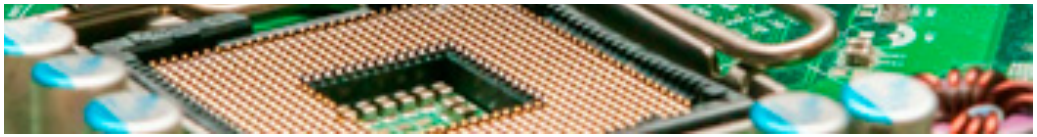
  rotate(leftarm, 45)

- Achievement goals: subobjetivos que deben ser alcanzados para que el plan continúe su ejecución (si en lugar de emplear "!" se emplea "!!", el plan no suspenderá su ejecución).

  !!at(home); call(john) en lugar de !at(home); call(john).

- Test goals: Para recuperar información de la BB o verificar si el agente cree algo.

  ?coords(Tarjet,X,Y).



Plans - Cuerpo

Mental Notes: Anotación *source(self)*. Sirven para añadir, modificar o eliminar nuevas creencias.

+currenttargets(NumTargets); [Se añade]

-+currenttargets(NumTargets); [Se modifica]

InternalActions: Son acciones que no modifican el entorno. Se diferencian de acciones del entorno por el carácter ".".

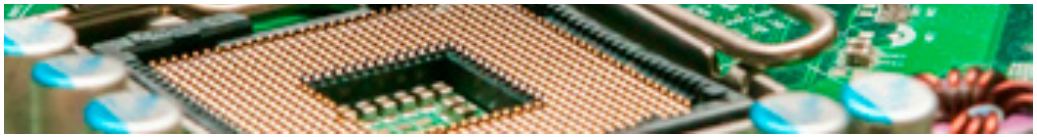
.print(...); .send(...);

Expressions: Su sintaxis es semejante a la de Prolog.

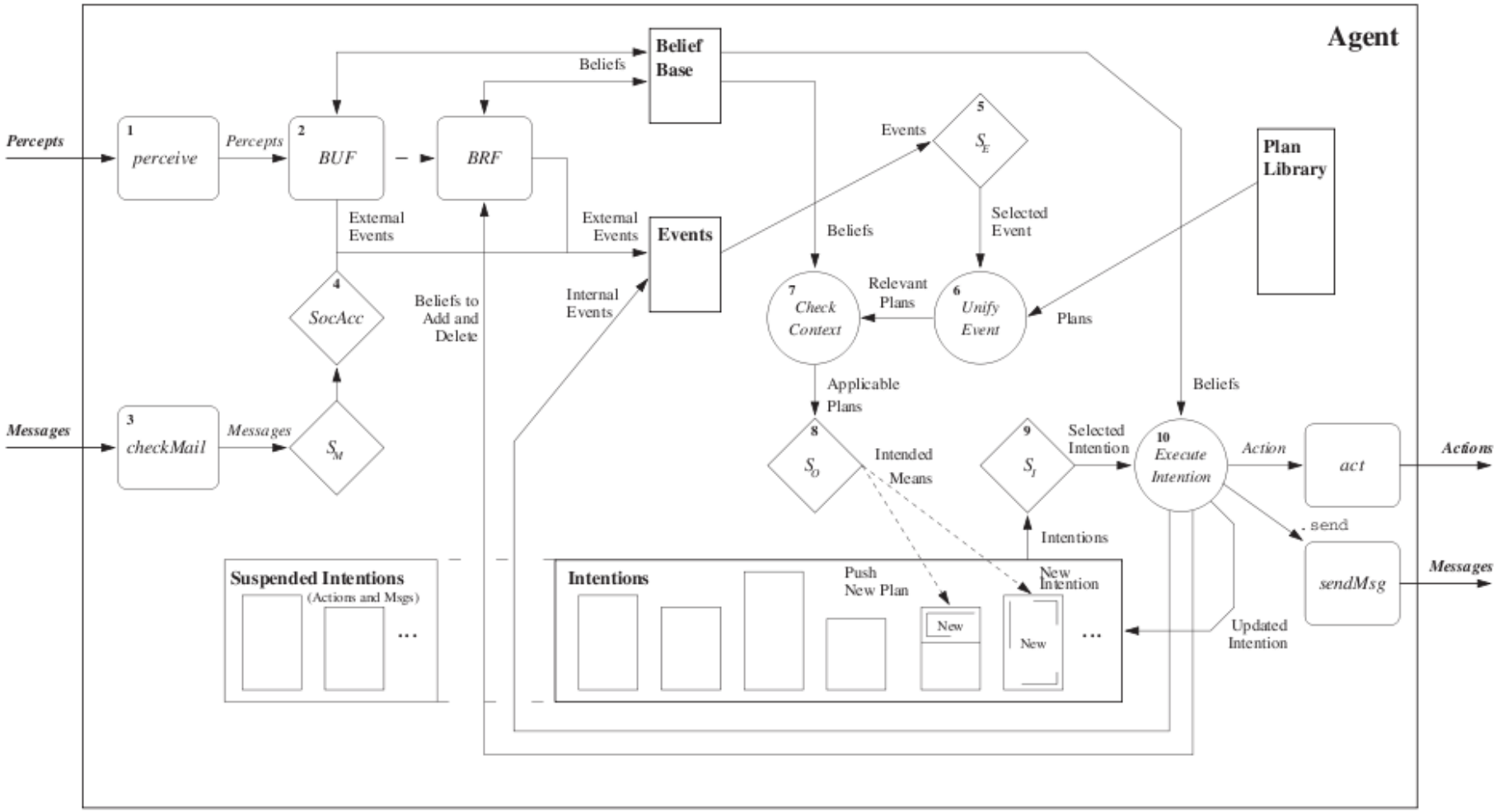
X >=Y*2

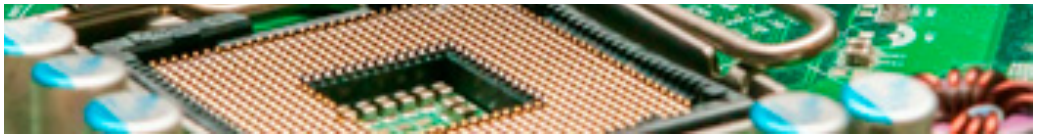
Plan Labels: Los planes pueden estar etiquetados

@labelte: ctxt<–body.



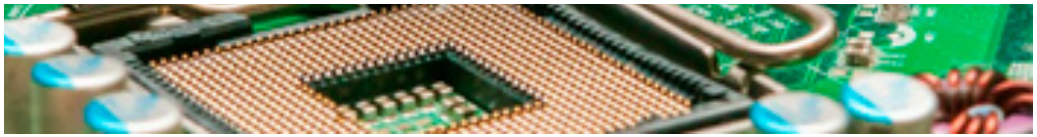
Arquitectura de Agente





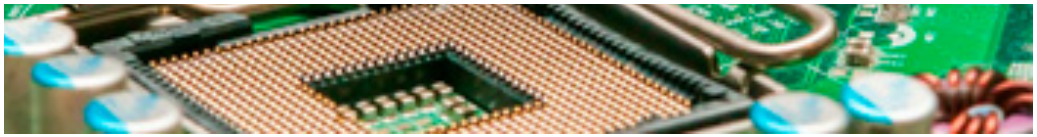
Fallo de un Plan

- Un plan puede fallar por tres causas principales:
  - Falta de planes relevantes o aplicables para un 'achievement goal'
  - Fallo de un 'test goal'
  - Fallo de una acción
- Cuando un plan falla se genera un evento -!g(goal deletion event) si se generó por la adición de un 'achievement goal' o 'test goal'.
- El plan que se dispare por el fallo se añade a la pila de intenciones del plan que ha fallado.



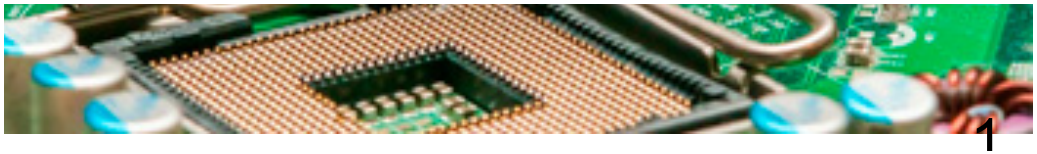
Comunicación

- Para cada creencia se anota su origen (id, self, percept)
- Cada agente posee una mailbox M y una función de selección de mensajes de M
- Performativas:
  - tell
  - untell
  - achive
  - unachive
  - tellHow
  - untellHow
  - askIf
  - askAll
  - askHow
- Ej: send(owner,tell,msg(M))



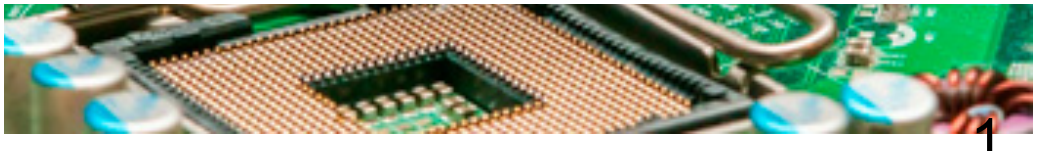
Ej. 1 Hello World

```
/* Creencias iniciales */
inicio.
/* Planes */
+inicio <- .print("Hola Mundo").    //plan que se dispara al inicio
```

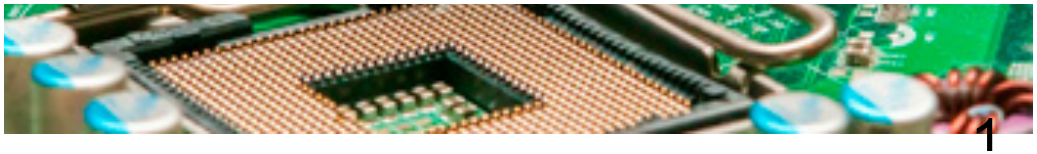
Ej 2. Factorial

```
/* Creencias iniciales */

fact(0,1).


/* Planes */

+fact(X,Y) : X < 5

  <-   +fact(X+1,  (X+1) * Y ).


+fact(X,Y) : X = 5

  <-   .print("fact 5 == ", Y ).
```

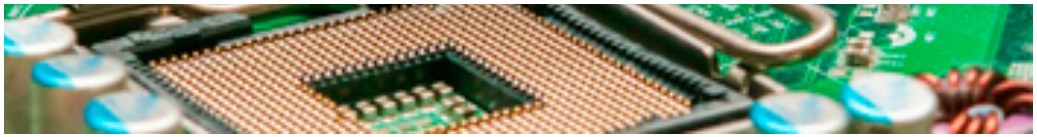
Ej 3. Paro y Arranque

```
/* Objetivos iniciales */
!dots.
!control.
/* Planes */
+!dots
  <- .print("."); // imprime puntos en un bucle sin fin
     !!dots.
+!control
  <- .wait(30); // este plan es un bucle que activa y desactiva el otro plan
     .suspend(dots); // suspende la intención asociada al plan dots
     .println;
     .wait(200);
     .resume(dots);  // reactiva la intención asociada al plan dots
     !!control.
```

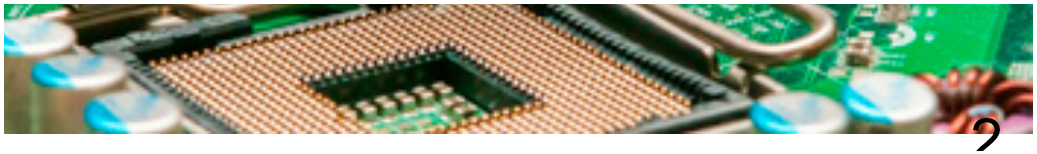
Ej 4. Mundo de Bloques (I)

```
// Un agente que soluciona el problema del mundo de bloques


/* Creencias iniciales y reglas */
clear(table).
clear(X) :- not(on(_,X)).  // la creencia clear(X) es cierta cuando no tiene nada encima
tower([X]) :- on(X,table). // la torre X es cierta cuando X está sobre la mesa
// X puede ser un bloque o varios
tower([X,Y|T]) :- on(X,Y) & tower([Y|T]).
// la torre va creciendo si un bloque X se pone sobre el bloque Y


/* Objetivos iniciales */
// Marca el estado final a alcanzar
!state([[a,e,b],[f,d,c],[g]]).
```

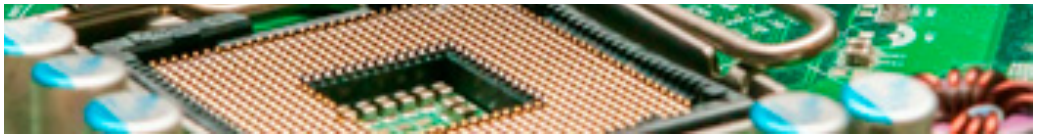
Ej 4. Mundo de Bloques (II)

```
/* Planes */

// Alcanzar una torre
+!state([])    <- .print("Finalizado!").
+!state([H|T]) <- !tower(H); !state(T).

// Alcanzar un estado donde la torre esté construida
+!tower(T) : tower(T). // La torre deseada ya existe, no hay nada que hacer
+!tower([T])     <- !on(T,table). // La torre es de un solo elemento
+!tower([X,Y|T]) <- !tower([Y|T]); !on(X,Y). // divido el problema en subproblemas
```

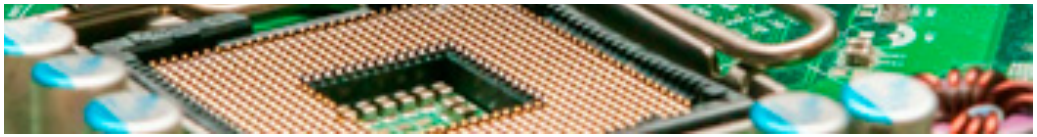
Ej 4. Mundo de Bloques (III)

```
// Planes para alcanzar la creencia de que un bloque X está encima de otro Y
+!on(X,Y) : on(X,Y). // ya conseguido
+!on(X,Y) <- !clear(X); !clear(Y); move(X,Y). // consigue que estén libres y mueve X sobre Y

// Planes para alcanzar la creencia de que un bloque X está libre
+!clear(X) : clear(X). // ya conseguido
// Quita el bloque que está encima cuando se necesita dejar libre el que está debajo X
+!clear(X)
  :  tower([H|T]) & .member(X,T)
  <- move(H,table);
     !clear(X).  // vuelve a lanzar el objetivo hasta que se consiga
```

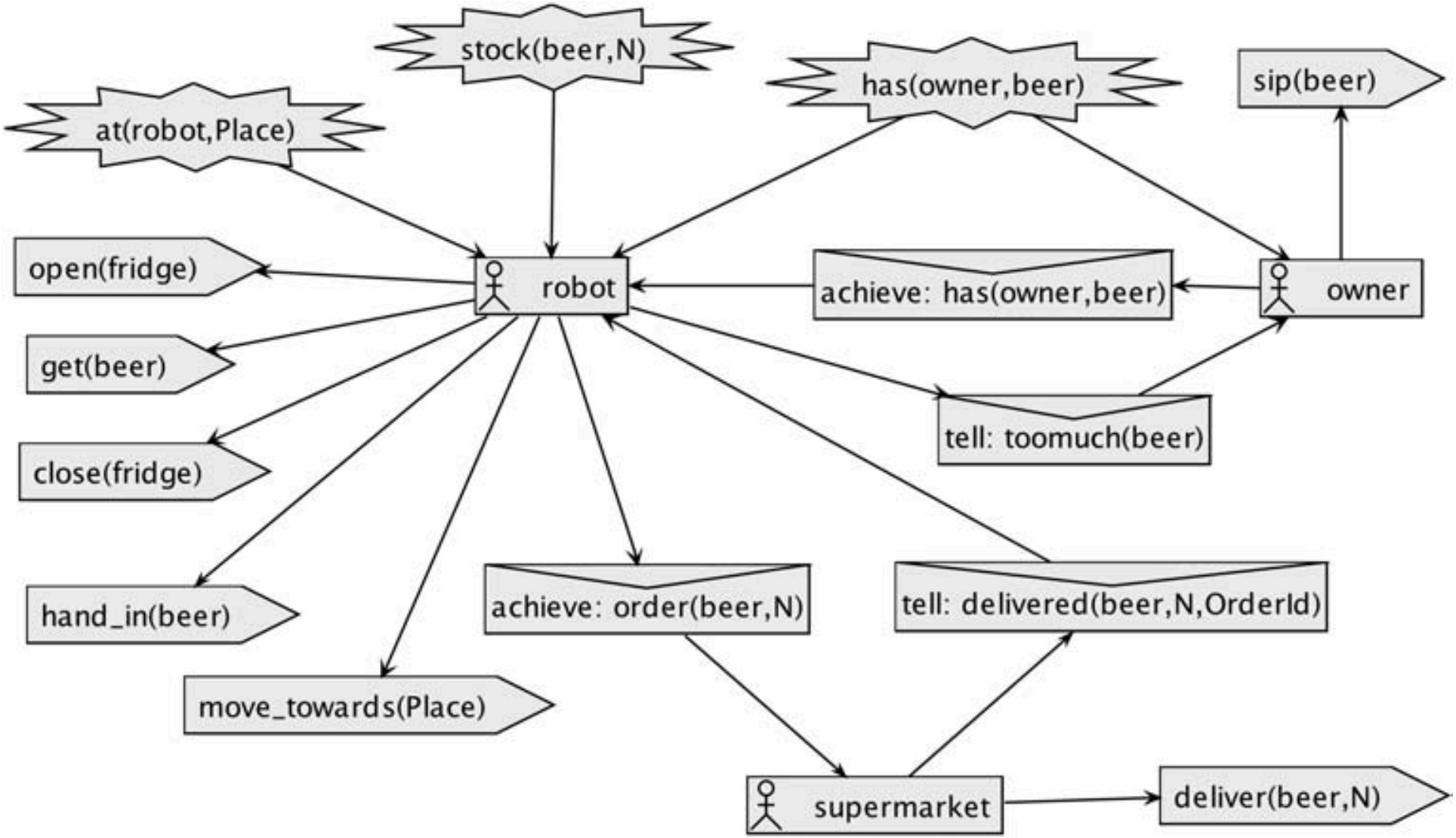



Un programa completo

A domestic robot has the goal of serving beer to its owner. Its mission is quite simple, it just receives some beer requests from the owner, goes to the fridge, takes out a bottle of beer, and brings it back to the owner. However, the robot should also be concerned with the beer stock (and eventually order more beer using the supermarket's home delivery service) and some rules hard-wired into the robot by the Department of Health (in this example this rule defines the limit of daily beer consumption).

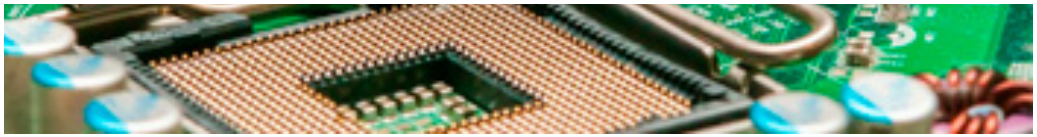


Escenario


stock(beer,N)
has(owner,beer)
sip(beer)
at(robot,Place)
open(fridge)
robot
achieve: has(owner,beer)
owner
get(beer)
tell: toomuch(beer)
close(fridge)
hand_in(beer)
achieve: order(beer,N)
tell: delivered(beer,N,OrderId)
move_towards(Place)
supermarket
deliver(beer,N)

Percepciones

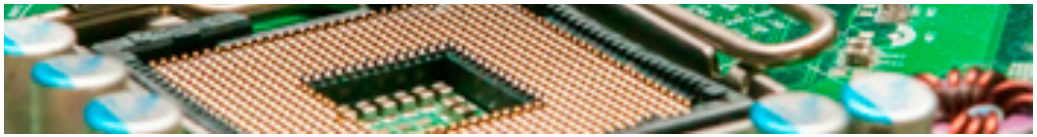
- at(robot,Place): to simplify the example, only two places are perceived, fridge (when the robot is in front of the fridge) and owner (when the robot is next to the owner). Thus, depending on its location in the house, the robot will perceive either at(robot,fridge) or at(robot,owner), or of course no at percept at all (in case it is in neither of those places);



Percepciones

- stock(beer,N): when the fridge is open, the robot will perceive how many beers are stored in the fridge (the quantity is represented by the variable N);

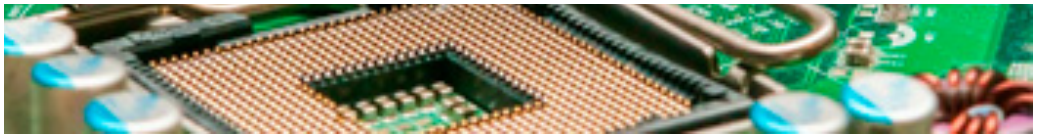
- has(owner,beer): is perceived by the robot and the owner when the owner has a (non-empty) bottle of beer.



Agente Propietario

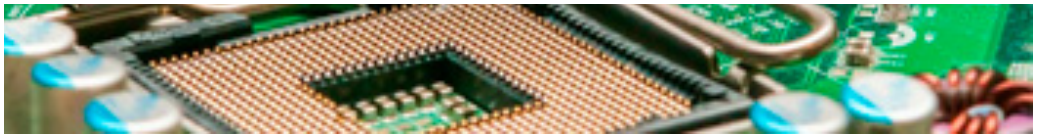
```
!get(beer). // initial goal

/* Plans */

@g

+!get(beer) : true

        <- .send(robot, achieve,
has(owner,beer)).

@h1

+has(owner,beer) : true

        <- !drink(beer).

@h2

-has(owner,beer) : true

        <- !get(beer).
```

```
// while I have beer, sip
@d1
+!drink(beer) : has(owner,beer)
        <- sip(beer);
           !drink(beer).
@d2
+!drink(beer) : not has(owner,beer)
        <- true.


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

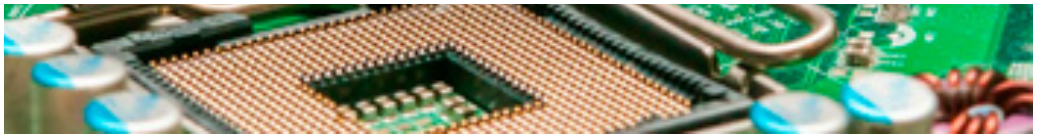
Agente Supermercado

```
last_order_id(1). // initial belief
// plan to achieve the the goal "order" from 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)).
```

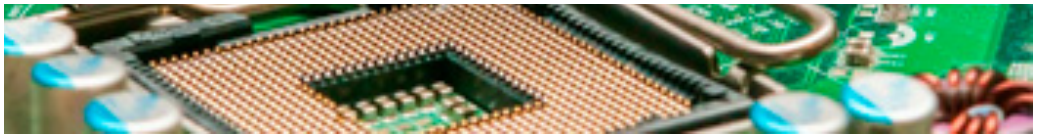
Agente Robot

```
/* Initial beliefs */
// initially, I believe that there are some beers in the fridge
available(beer,fridge).
// my owner should not consume more than 10 beers a day :-)
limit(beer,10).
/* Rules */
too_much(B) :-
        .date(YY,MM,DD) &
        .count(consumed(YY,MM,DD,_,_,_,B),QtdB) &
        limit(B,Limit) &
        QtdB > Limit.
```

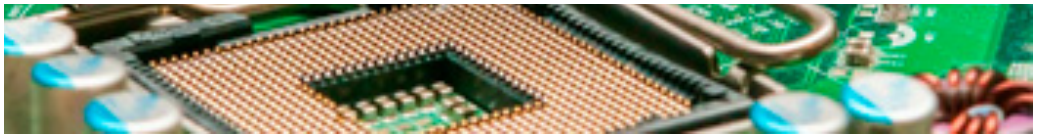
Agente Robot (II)

```
/* Plans */
@h1
+!has(owner,beer)
        : available(beer,fridge) & not too_much(beer)
        <- !at(robot,fridge);
                open(fridge);
                get(beer);
                close(fridge);
                !at(robot,owner);
                hand_in(beer);
                // remember that another beer will be consumed
                .date(YY,MM,DD); .time(HH,NN,SS);
                +consumed(YY,MM,DD,HH,NN,SS,beer).
```

Agente Robot (III)

```
@h2
+!has(owner,beer)
        : not available(beer,fridge)
        <- .send(supermarket, achieve, order(beer,5));
                !at(robot,fridge). // go to fridge and wait there.

@h3
+!has(owner,beer)
        : too_much(beer) & limit(beer,L)
        <- .concat("The Department of Health does not allow me ",
                                "to give you more than ", L,
                                " beers a day! I am very sorry about that!",M);
                .send(owner,tell,msg(M)).
```

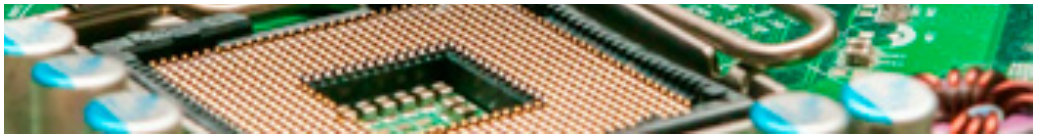
Agente Robot (IV)

- @m1

```
+!at(robot,P) : at(robot,P) <- true.
```

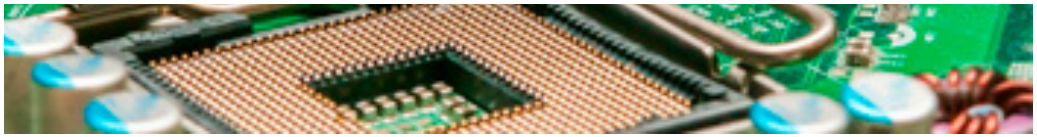
- @m2

```
+!at(robot,P) : not at(robot,P)
                <- move_towards(P);
                   !at(robot,P).
```

Agente Robot (V)

```
// when the supermarket finishes the order, try the 'has' goal
again
@a1
+delivered(beer,Qtd,OrderId)[source(supermarket)] : true
                <- +available(beer,fridge);
                   !has(owner,beer).
```

Agente Robot (VI)

```
// when the fridge is opened, the beer stock is perceived and thus
the available belief is updated
@a2
+stock(beer,0)
        : available(beer,fridge)
        <- -available(beer,fridge).
@a3
+stock(beer,N)
        : N > 0 & not available(beer,fridge)
        <- +available(beer,fridge).
```