



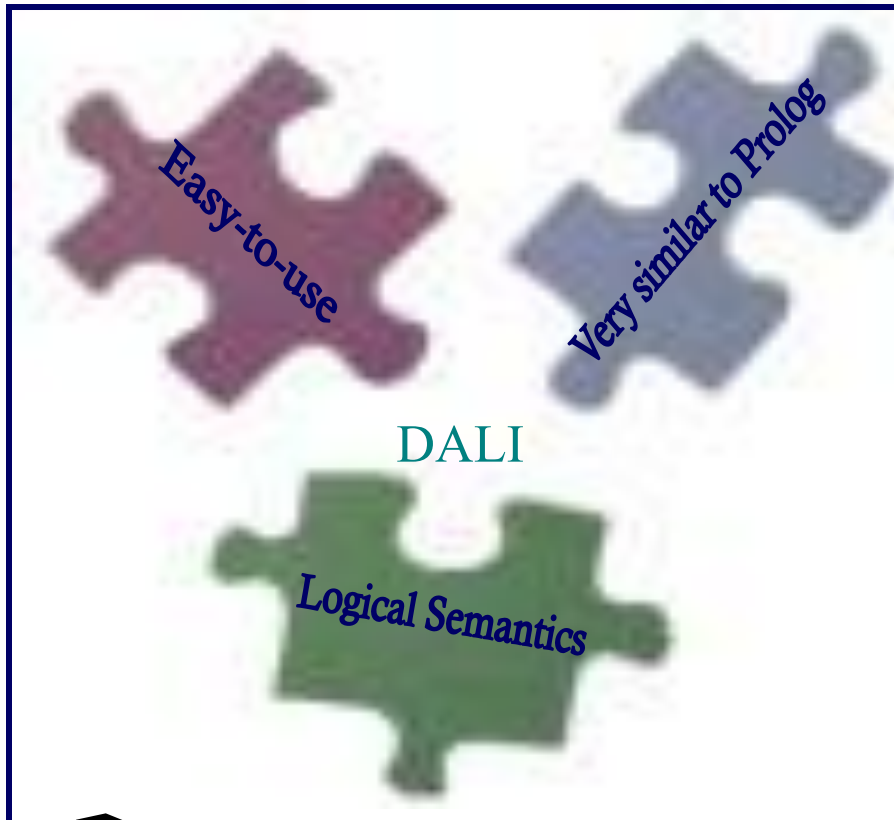
The DALI Logic Programming Language: an Overview

Stefania Costantini
Arianna Tocchio

Dip. Di Informatica, Univ. degli Studi di L'Aquila
Stefania.costantini@univaq.it

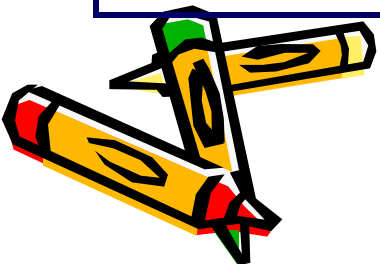


DALI: a Logic Programming language for Agents and Multi-Agent Systems



Prolog for Agents:

- different classes of events and their interaction;
- the interleaving of different activities;
- a concept of time.



DALI is a general-purpose language



- Supports the agent-oriented paradigm
- What you may wish to implement yourself:
 - An observe-think-act cycle
 - An Agent Architecture
 - A model of cooperation
 - ... and much more



Declarative Semantics (overview)

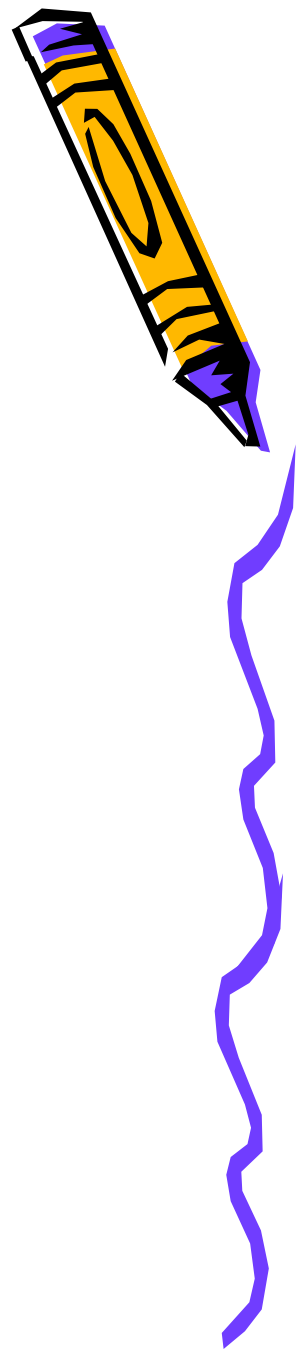


- P_{Ag} transformed into P_{HAg} (pure Horn)
- Events affect P_{HAg}
 - Program Evolution Sequence $PE \equiv [P_0, \dots, P_n]$
 - Model Evolution Sequence $[M_0, \dots, M_n]$ with M_i is the model of P_i
- We can use model-checking for verifying properties



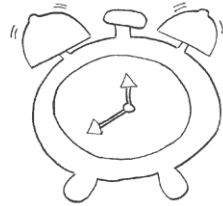
Procedural Semantics (Computational Model)

- Extended Resolution Procedure
 - That interleaves different activities
 - That can be tuned by the user via directives



DALI: Perspectives on External Events

alarm_signal



Stage 1: event perceived, but no reaction yet.

Function: reasoning about what's happening.

Notation: *present event*, written *alarm_signalN*

Stage 2: reaction to the event

Function: triggering an activity according to the event.

Notation: *external event*, written *alarm_signalE*

Stage 3: after reaction, the agent is able to remember the event.

Function: reasoning about what happened in the past.

Notation: *past event*, written *alarm_signalP*



Events and Actions

Distinction between reasoning about events and reacting to events

visitor_arrived :- bell_ringsN.

bell_ringsE :> open_doorA.

reactive rules *evE:> Reaction*

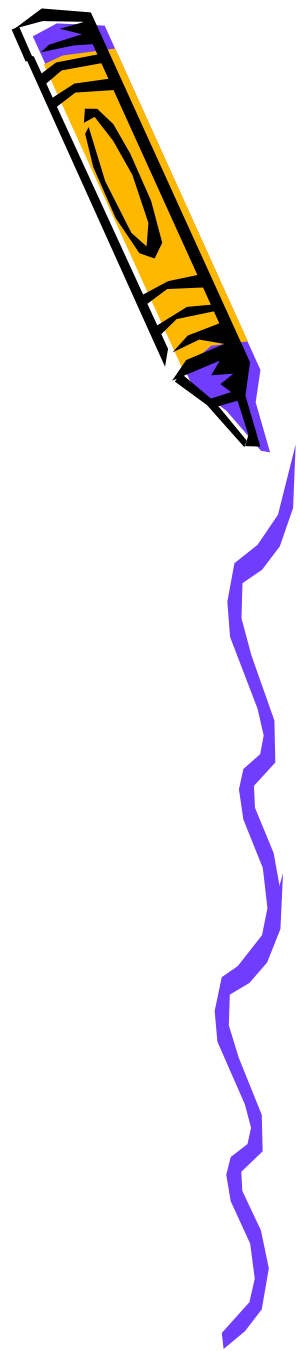
After reaction, the event becomes a past event *evP*

Actions with or without preconditions:

open_doorA:< have_key.

open_doorA:< not have_key, get_key.

Action *actA* becomes a past action *actP*



More about External events

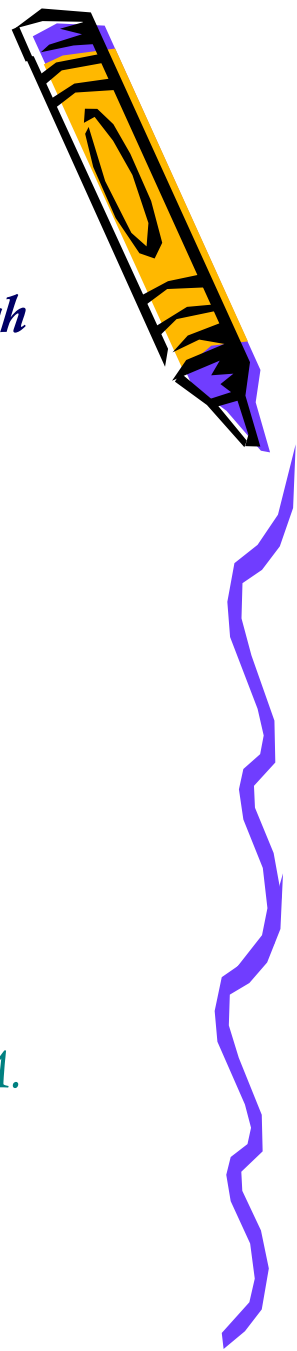
In the implementation, events are time - stamped, and the order in which they are “consumed” corresponds to the arrival order.

The time - stamp can be useful for introducing into the language some (limited) possibility of reasoning about time.

alarmE(K) : T :> alarmP(K):T1, T1-T < threshold, alert_operatorA.

Conjunction of events in the head of a reactive rule
(must occur within an amount of time, set by a directive):

alarmE(K1), alarmE(K2) :> K1 != K2, close_accessA , alert_operatorA.



Past Events and Actions: Memory

ACCESS CONTROLLER (to a system, a place, a device, ecc.)

situation_okE :> open_accessA.

situation_unsafeE:> limit_accessA.

open_accessA:< access_is_limited.

access_is_limited :- limit_accessP.

limit_accessA :< access_is_open.

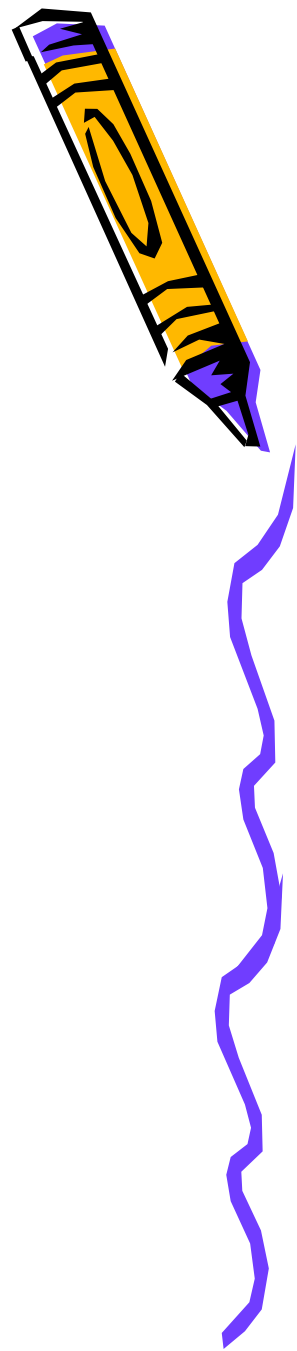
access_is_open :- open_accessP.

Past events/actions are kept according to directives.

Amount of time /forever/terminating condition:

keep open_accessP until limit_accessA.

keep limit_accessP until open_accessA.

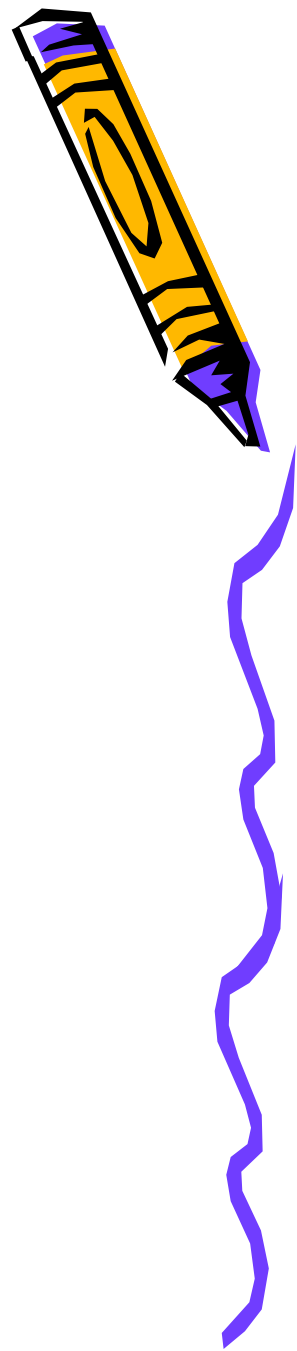


Internal Events: Proactivity

*An internal event is a conclusion reached by the agent,
INTERPRETED AS AN EVENT
to which the agent reacts similarly to the external ones!*

Role: internal conclusions initiate further inference
(proactivity!).

Internal events model maintenance goals



Internal Events: Proactivity

Internal conclusions initiate further inference:

*ready(cake) :- in_the_ovenP(cake), color(cake,golden),
smell(cake,good).*

*readyI(cake):>
take_from_ovenA(cake),switch_off_ovenA,
eatA(cake).*

Also internal events become past events



Internal Events: Proactivity

Conclusions corresponding to internal events are automatically attempted from time to time (default or set by directives).



```
attack_happening(Cond) :- suspicious(Cond),  
                           probability(Cond,attack) > threshold.  
attack_happeningI(Cond) := start_emergencyA(Cond),  
                           alert_operatorA.
```



Internal Events: Proactivity

*User Directives for Internal Events
(conceptually, actual syntax is different):*

*try Int_Ev since [Time / Cond] [unless Exception] until [Time / Cond]
[frequency F]*

try Int_Ev forever [frequency F]

priority E1, E2, ..., En % presently implemented: priority E high



Special internal Events: Goals



- *Goals start being attempted*
 - *when encountered during the inference process*
 - *or when invoked by an external event.*
- *Each goal pG will be automatically attempted until it succeeds, and then expires.*



Special internal Events: Goals



- *If multiple definitions of a goal pG are available:*
 - *they are (as usual) applied one by one by backtracking,*
 - *success of one alternative prevents any further attempt.*
- *On success, reactive rule $pI :> \text{Body}$ is applied.*



Uses of Goals



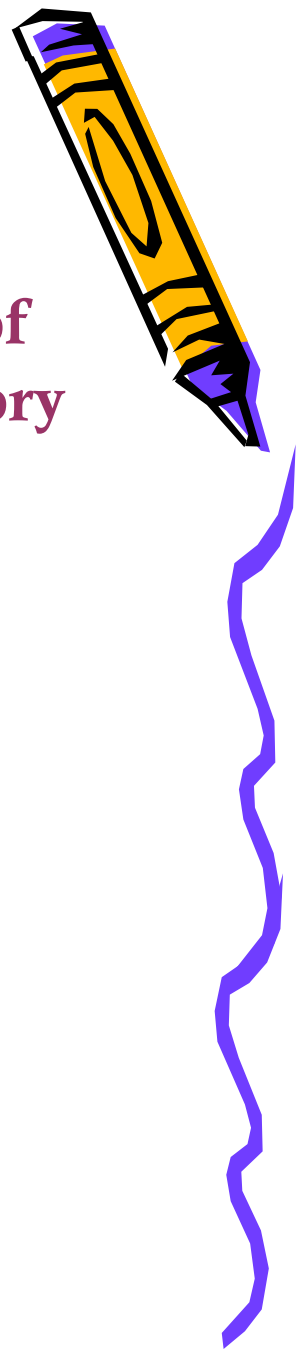
- *Perform simple planning tasks.*
- *Perform complex planning:*
 - *Invoke an Answer Set Solver for obtaining alternative plan*
 - *Selecting and applying a plan*
 - *Selecting an alternative plan on failure*
 - *Re-plan if needed*



User directives

Activating a DALI agent involves the pre-processing of directives about how to cope with events and memory

- *Defining events (predicates declared as external/internal events)*
- *Priorities among external and internal events*
- *Time slot for considering two events simultaneous*
- *Frequency for attempting internal events*
 - check try_to_connect frequency 5s
- *Terminating conditions for attempting internal events*
 - keep try_to_connect until connection_established
 - keep try_to_connect until 5:00pm
 - keep try_to_connect unless network_failure
- *How long to keep past events and actions*
 - keep open_the_doorP until close_the_doorP.
 - keep its_summertimeP until Sept 21
 - keep bornP(daniele, date(Aug,27,1993)) forever



Procedural Semantics

Procedural semantics of DALI: extension to SLD - resolution.

A DALI goal is a disjunction $G^1 ; G^2 ; \dots ; G^n$

The procedural behavior of a DALI agent consists of the interleaving of the following steps.

1. Resolve a sub-goal like in plain Horn - clause language.
2. Responding to either external or internal events. The interpreter picks up either an external event from list EV or an internal event from list IV, and adds this event G^{ev} as a new subgoal.

Thus, goal $G^1 ; G^2 ; \dots ; G^n$ becomes $G^1 ; G^2 ; \dots ; G^n ; G^{ev}$
and G^{ev} is inserted into list PV of past events.

3. Trying to prove a goal corresponding to an internal event. The interpreter picks up an atom from list EVT, and adds this atom G^{evt} as a new subgoal. Thus, goal $G^1 ; G^2 ; \dots ; G^n$ becomes $G^1 ; G^2 ; \dots ; G^n ; G^{evt}$



Declarative Semantics

Declarative semantics of DALI program P based on standard declarative semantics (**Least Herbrand Model**)

Starting point: modified program P_s , obtained from P by means of syntactic transformations that specify how the different classes of events are coped with.

P_s is the basis for the evolutionary semantics, that describes how the agent is affected by reception of events.



Agent evolution according to events

Program P_s must be actually affected by the events, by means of subsequent syntactic transformations.

Declarative semantics of agent program P at a certain stage:
declarative semantics of the version of P_s at that stage.

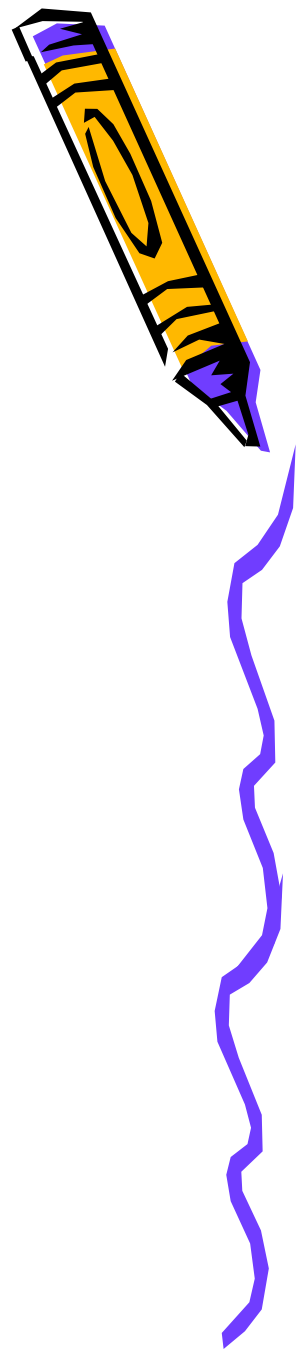
$P_0 \equiv \langle P_s, [] \rangle$ (initially no event has happened).

$P_n \equiv \langle Prog_n, Event_list_n \rangle$,

where $Event_list_n$ is the list of the n events that have happened,
and $Prog_n$ is the current program



P_n obtained from P_0 step by step by
means of a transition function Σ



Program Evolution

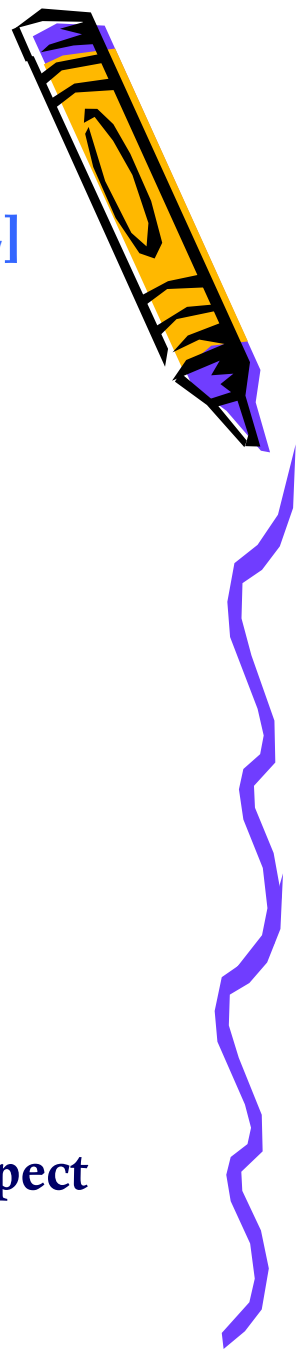
Definition 2 Let P_s be a DALI program, and $L = [E_n, \dots, E_1]$ be a list of events. Let $P_0 = \langle P_s, [] \rangle$ and $P_i = \Sigma(P_{i-1}, E_i)$. The list $PE \equiv [P_0, \dots, P_n]$ is the *Program Evolution* of P_s with respect to L (where $P_i \equiv \langle Prog_i, Event_list_i \rangle$).

Model Evolution

Definition 3 Let P_s be a DALI program, L be a list of events, and PL be the program evolution of P_s with respect to L . Let M_i be the Least Herbrand Model of $Prog_i$. The sequence $ME \equiv [M_0, \dots, M_n]$ is the *Model Evolution* of P_s with respect to L , and M_i the *instant model* at step i .

Evolutionary Semantics

The evolutionary semantics \mathcal{E}_{P_s} of P_s with respect to L is the couple $\langle PE, ME \rangle$.



In practice: Input to the Interpreter

The input to the DALI interpreter consists in a `<file>.txt` that contains:

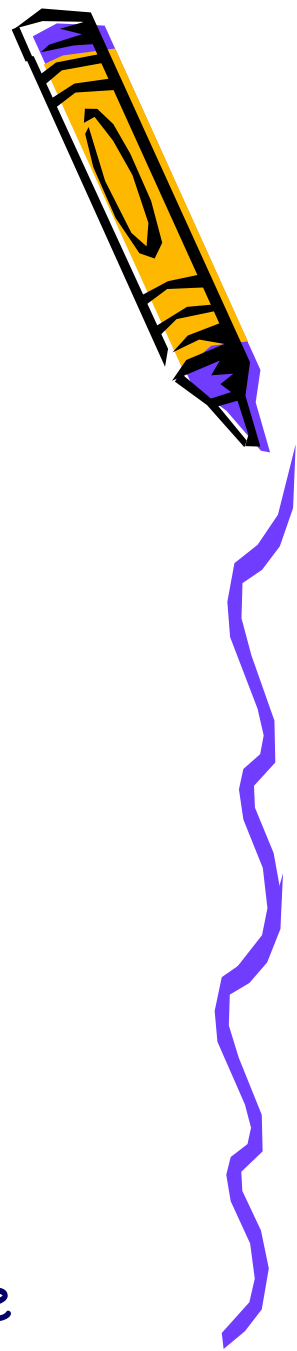
Mandatory

- The name `<name>` of the agent
- The program

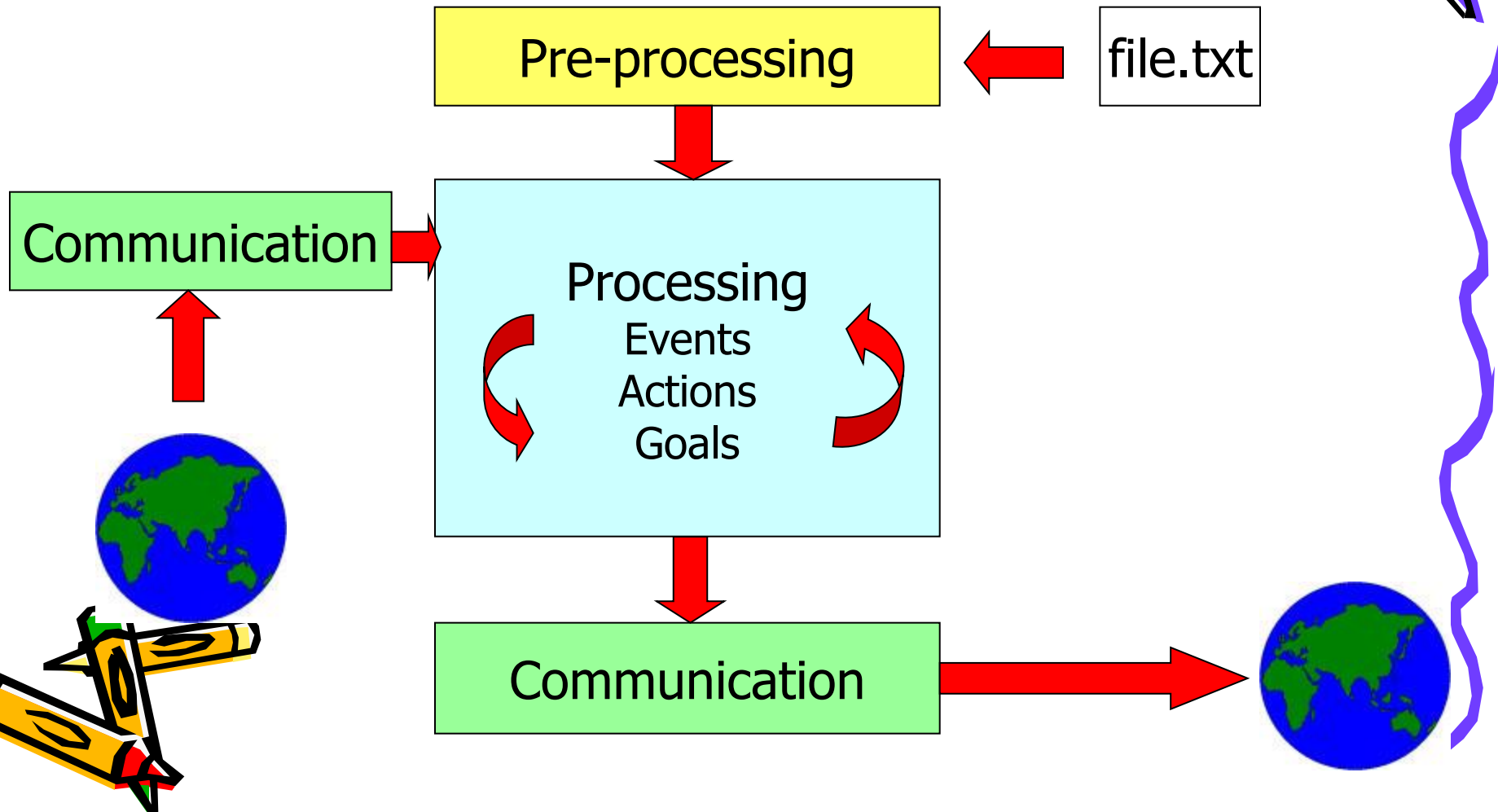
Optional

- The ontology she is meant to use
- The language (Italian/English...)
- The libraries she wants to import
 - `communication.txt`, `communication_fipa.pl`
 - `<name>.plf` for directives about events

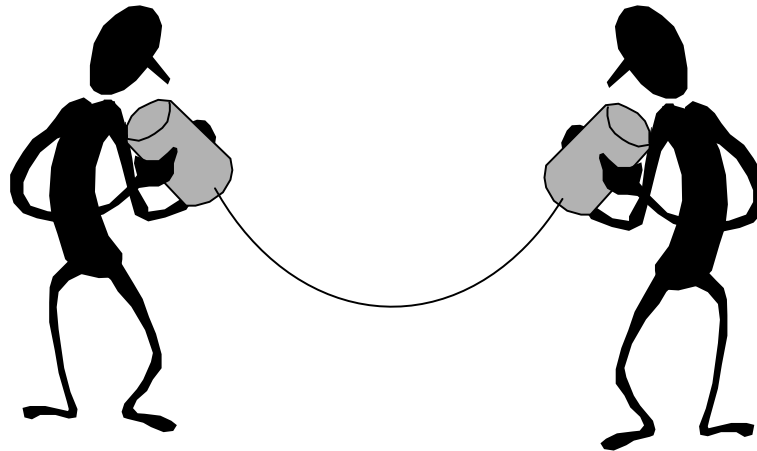
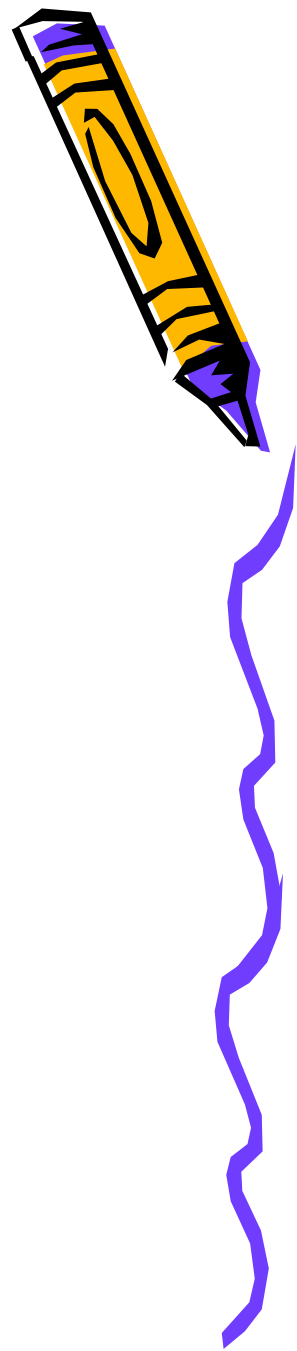
A syntax-driven editor for DALI is available



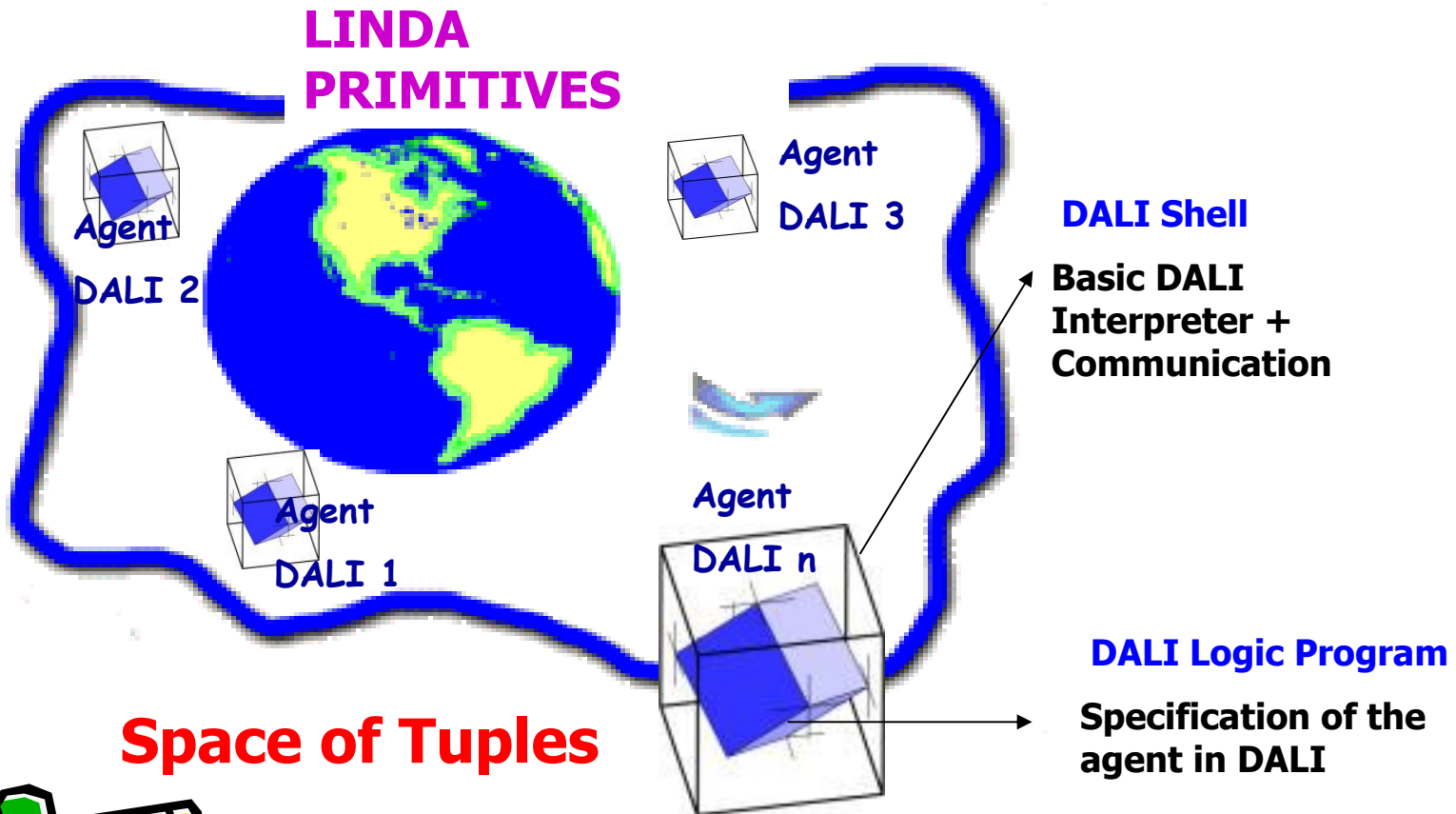
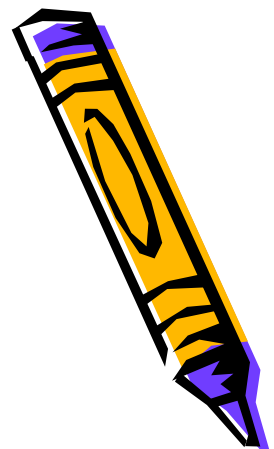
DALI + Communication: Overview



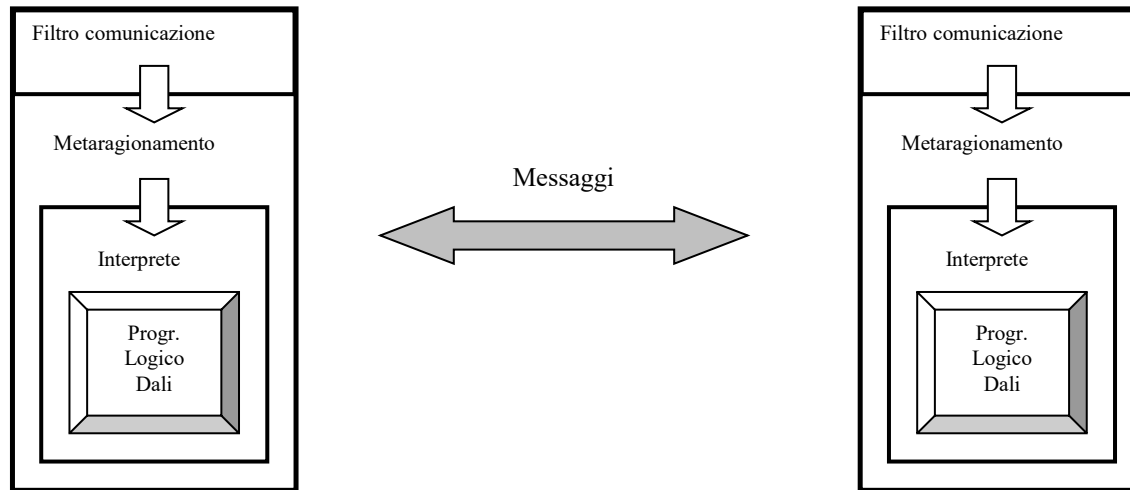
Communication in DALI



Communication: Basic Framework



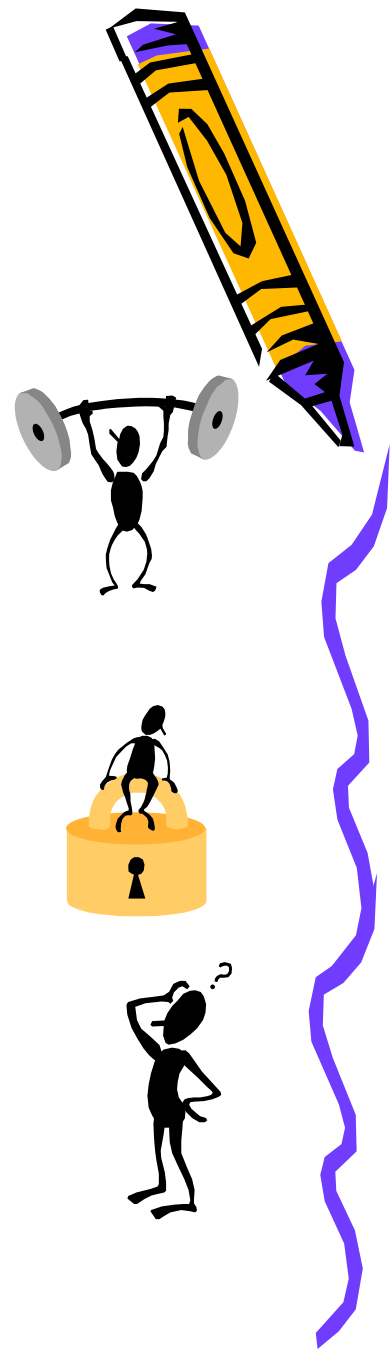
Communication Layers



- Communication filter and protocol definition;
- Meta-reasoning on messages (possibly exploiting ontologies);
- Interpreter, that implements FIPA primitives, plus others.

Objectives

- An agent should be equipped with its own communication protocol, tailored to the domain/application/situation at hand.
- An agent should be able to filter incoming messages according to its own policies, interests and objectives and in order to ensure its own safety.
- An agent should be able to cope with messages that are not immediately intelligible (by exploiting ontologies).



Communication

Communication:

- *an external event can be a message from another agent*
- *symmetrically, an action can consist in sending a message.*

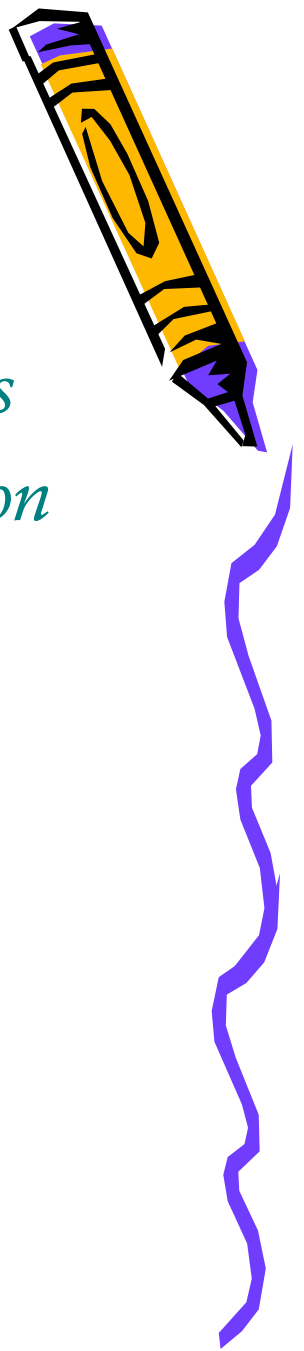
Message seen by the interpreter in the syntax:

*messageA(To,
Comm_primitive(From, Content, In_Reply_to[optional]), Timestamp)*



Communication

- *Comm_primitive = FIPA-compliant set + Others*
- *File communication.txt defining communication constraints/protocol, meta-understanding*
 - *Primitives tell/told for filtering*
 - *Procedure meta for understanding*



Examples of Communication Primitives



- FIPA primitives: aimed at influencing "mental state" of other agents. Examples:

`propose(Action, [cond1, ..., condn], A2)`
`accept_proposal(Action, [a_cond1..., a_condn], A1))`

- DALI additional primitives
 - "simple" communication:

`send_message(Event, A2)`

- Request for a service

`execute_proc(Proc, A2)`



How do agents defend themselves?...



- Communication may be unreliable, thus potentially resulting in damage of the receiver agent's knowledge.



- Other agents may (on purpose or not) send "wrong" messages. How to detect "wrong messages?"
- Sometimes messages are uninteresting. How to recognize "interesting" messages?

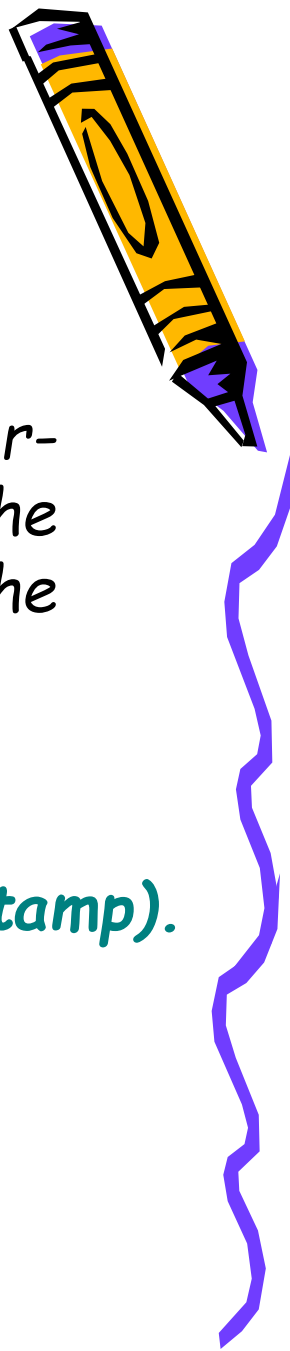


Constraints on Incoming Messages

Messages accepted only if they satisfy user-defined constraints (red part defined by the user, green part automatically added by the system).

```
told(Ag, Comm_primitive(...)):-  
  constraint1, constraint2, ..., constraintn,  
  message(To, Comm_primitive(Ag, ...), Timestamp).
```

Red: agent program Green: interpreter



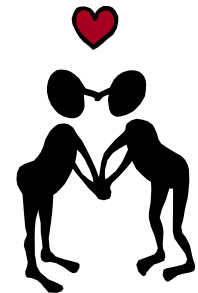
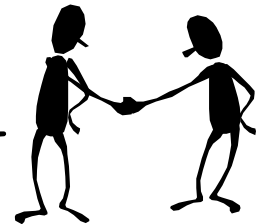
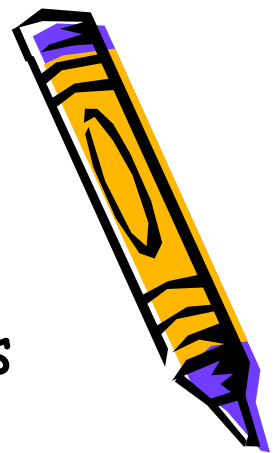
Constraints on Incoming Messages

EXAMPLES, a programmer can write her/his own rules

```
told(Sender_agent,send_message(Ext_event)):-  
not(unreliableP(Sender_agent)),  
interested_in(Ext_event).
```

```
told(Sender_agent,execute_proc(Procedure)):-  
not(unreliableP(Sender_agent)),  
defined(Procedure).
```

```
told(Sender_agent,query_ref(Proposition,3)):-  
not(unreliableP(Sender_agent)),  
nice(Sender_agent).
```





What if an agent does not understand a message?...

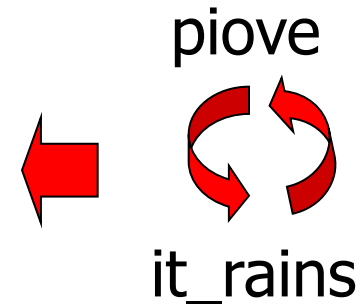
- The agent can try to exploit ontologies
 - General-purpose
 - Associated to the agent: domain-dependent and/or learnt.
- Meta-reasoning can help: commonsense reasoning, exploiting properties of relations.
- The DALI solution: meta-rules automatically applied on message contents if *told* fails in the first place.



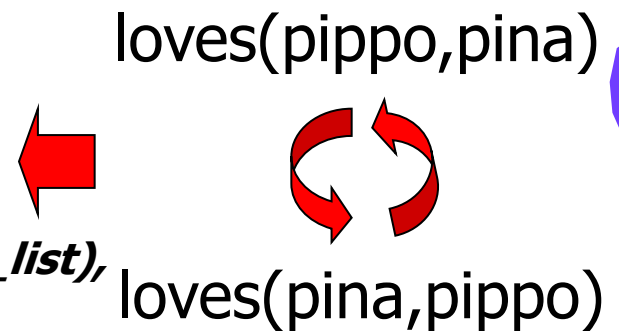
What is meta-reasoning up to...



```
meta(Initial_term,Final_term,Agent_Sender):-  
  clause(agent(Agent_Receiver),_),  
  functor(Initial_term,Funcor,Arity),Arity=0,  
  ((ontology(Agent_Sender,Funcor,Equivalent_term);  
   ontology(Agent_Sender,Equivalent_term,Funcor));  
   (ontology(Agent_Receiver,Funcor,Equivalent_term);  
    ontology(Agent_Receiver,Equivalent_term,Funcor))),  
  Final_term=Equivalent_term.
```



```
meta(Initial_term,Final_term,Agent_Sender):-  
  functor(Initial_term,Funcor,Arity),Arity=2,  
  symmetric(Funcor),Initial_term=..List,  
  delete(List,Funcor,Result_list),  
  reverse(Result_list,Reversed_list),  
  append([Funcor],Reversed_list,Final_list),  
  Final_term=..Final_list.
```

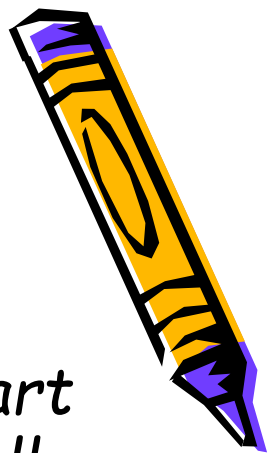


Implicitly...

How things work (declaratively): red part defined by the user, green part automatically added by the system.

```
told(Ag, Comm_primitive(Content1)):-  
[constraint1, constraint2, ..., constraintn],  
message(To, Comm_primitive(Ag, ...), Timestamp),  
meta(Content, Content1, Ag).
```

Declarative Semantics: based on Implicit Reflection

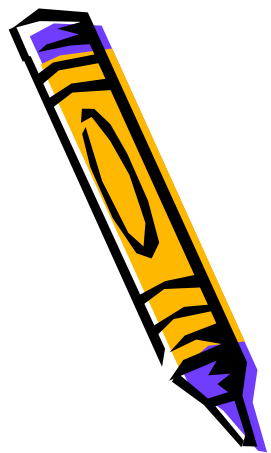


Constraints and Condition on Outcoming Messages

```
tell(To, From, Comm_primitive(...)):-  
    condition1, ..., conditionm,  
    constraint1, ..., constraintn.
```

```
tell(To, _, send_message(...)):-  
    told(To, k), not(enemy(To)).
```

Combinations tell/told: protocols



Cooperation and Trust

The patient gets ill and speaks to the family doctor about his symptoms...

i_am_ill



coldP, bone_painP, high_temperatureP

coughP, thoracic_painP, difficult_breathP, high_temperatureP.

`send_message(i_am_ill(patient), patient)`

`send_message(take_antibiotic, doctor)`

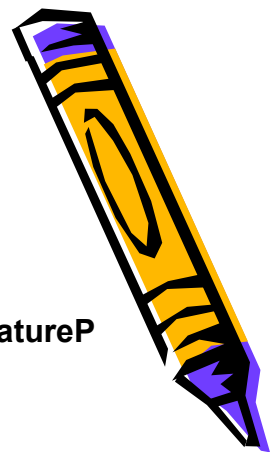
`send_message(consult_lung_doctor, doctor))`



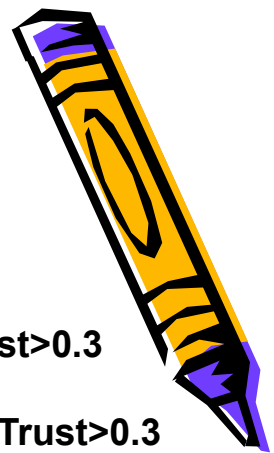
The patient, by consulting the yellow pages, finds two lung doctors:

lung_doctor₁ and lung_doctor₂

He asks some friends about the doctors...



Cooperation and Trust



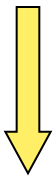
trust(patient,friend_nurse,0.5)
trust(patient,friend_clerk,0.4)



tell(To,patient,Primitive(...):- trust(patient,To,Trust),Trust>0.3

told(Sender,Primitive(...)):- trust(patient,Sender,Trust),Trust>0.3

send_message(what_about(lung_doctorX,
,patient),patient))



send_message(what_about(lung_doctor_x,
patient),patient))



inform(ability(lung_doctor₁,0.7),friend_nurse)

inform(ability(lung_doctor₂,0.3),friend_nurse)

inform(ability(lung_doctor₁,0.4),friend_clerk)

inform(ability(lung_doctor₂,0.8),friend_clerk)



friend_nurse



friend_clerk

ability(lung_doctor₁,0.7)
ability(lung_doctor₂,0.3)

ability(lung_doctor₁,0.4)
ability(lung_doctor₂,0.8)



Cooperation and Trust

```
trust(patient,friend_nurse,0.4)
trust(patient,friend_clerk,0.3)
```



The trust in the friend nurse is higher.
I choose lung_doctor₁.

After one week...

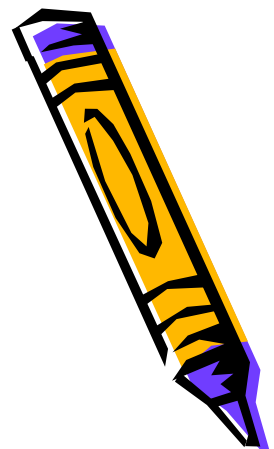
```
recovered_by(Lung_doctor):- go_to_lung_doctorP(Lung_doctor),not(high_temperatureP),
                             not(thoracic_painP),not(coughP),not(difficult_breathP).
recovered_byI(Lung_doctor):> recommended_from(Friend),increment_trust_toA(Friend,0.2),
                             assert(trust(patient,Lung_doctor,0.5)).
```

➡ trust(patient,friend_nurse,0.6)

```
not_recovered_by(Lung_doctor):- go_to_lung_doctorP(Lung_doctor),high_temperatureP.
not_recovered_by(Lung_doctor):- go_to_lung_doctorP(Lung_doctor), thoracic_painP.
not_recovered_by(Lung_doctor):- go_to_lung_doctorP(Lung_doctor), coughP.
not_recovered_by(Lung_doctor):- go_to_lung_doctorP(Lung_doctor), difficult_breathP.
not_recovered_byI(Lung_doctor):> recommended_from(Friend),
                                decrement_trust_toA(Friend,0.2),
                                assert(trust(patient,Lung_doctor,0)).
```



➡ trust(patient,friend_nurse,0.2)




Level of Trust influences communication and choices.

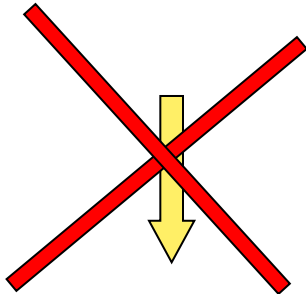
In fact, if the patient after a week is still ill...



`trust(patient,friend_nurse,0.2)`
`trust(patient,friend_clerk,0.3)`



`told(Sender,Primitive(...)):- trust(patient,Sender,Trust),Trust>0.3`
`tell(To,patient,Primitive(...)):- trust(patient,To,Trust),Trust>0.3`



`send_message(what_about_ability(lung_doctorx,patient),patient))`

The filter-level blocks the communication for friend_nurse



friend_nurse

`inform(ability(lung_doctor1,0.4),friend_clerk)`
`inform(ability(lung_doctor2,0.8),friend_clerk)`

The patient will now choose lung_doctor₂!



friend_clerk





The End!

