

CARA (Contextual Agent Robot Architecture)

a joint research of

- DEIS-University of Bologna

E.Denti, A.Natali, A.Omicini

CNR Project on New Programming Languages

- DII-University of Parma

F.Zanichelli, S.Caselli

CNR Project on Robotics

experimenting *extended* LP in a *real, advanced* application

Experimenting *extended* LP in a *real, advanced* application

Extended : conventional LP (Prolog) is not adequate for real programming

- Logic programming promotes a new style of software design and development
- LP reduces the gap between specification and coding
- Predicates, unification and deduction subsume traditional concepts such as procedure, assignment, control
- Logic programs can be interpreted as networks of parallel processes
- Innovative meta-programming techniques are possible

BUT

LP is inadequate to face *real* needs of software design and development

Real : robot programming presents most of the concrete problems of industrial software design and development

- general software engineering issues
- need of advanced programming environments
- need of explicit control and real-time

while demanding **advanced** features to integrate

ROBOT-LEVEL PROGRAMMING

(for real-time event handling and basic motion control)

TASK-LEVEL PROGRAMMING

(for high-level, "intelligent" control)

HOT TOPIC

Reactive vs. planning-based systems

THE LIMITS OF LOGIC PROGRAMMING

- Programming in the large

Procedures provide a too fine degree of modularity

Modules (as syntactic containers) are not adequate

Objects should not be an "added-on"

- The control issue

Separation between knowledge and control is nice, but several applications require *control over control*

THE CONSTRAINT

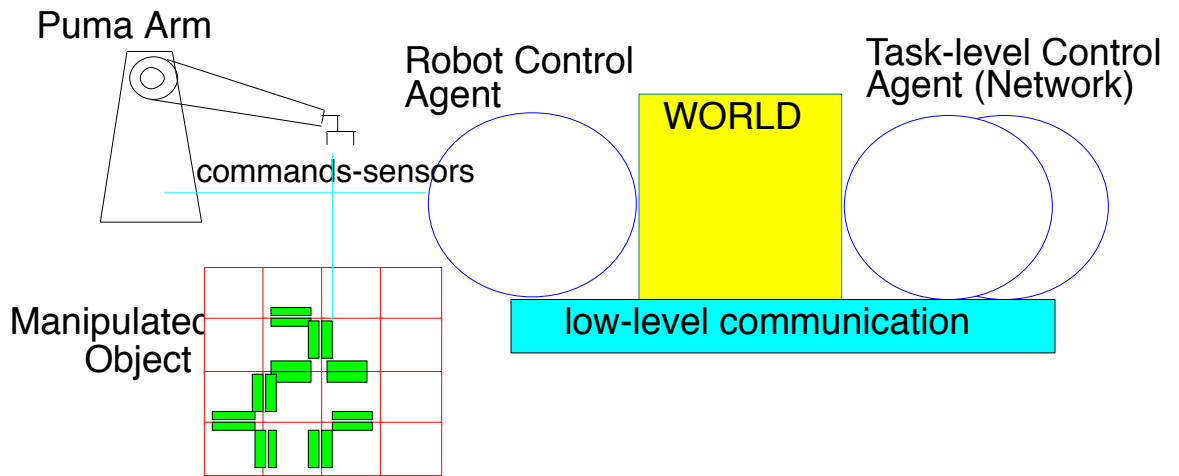
- Any extension should provide useful concepts, abstractions and mechanisms without undermine the logic foundations of LP

EXTENSIONS TO LP IN CARA

aim at providing an advanced support for robot control

- At architectural level
 - Provide an environment for the design and development of different kinds of robot architecture (planning-based, hybrid, purely reactive)
- At software development level
 - Provide new abstractions and open, extendible tools
 - Support *spiral models* for software design and development
 - Facilitate the transition from a simulated (testing) environment to the real one
- At system programming level
 - Provide a bridge (an unifying framework) between knowledge-based subsystems and low-level (real-time) control features (inter-agent communication and control)

CARA Architecture



Robot Control Agent (a server)

a RCCL-based process performing (hard)real-time operations (robot basic moves)

Task-level Agent (the master)

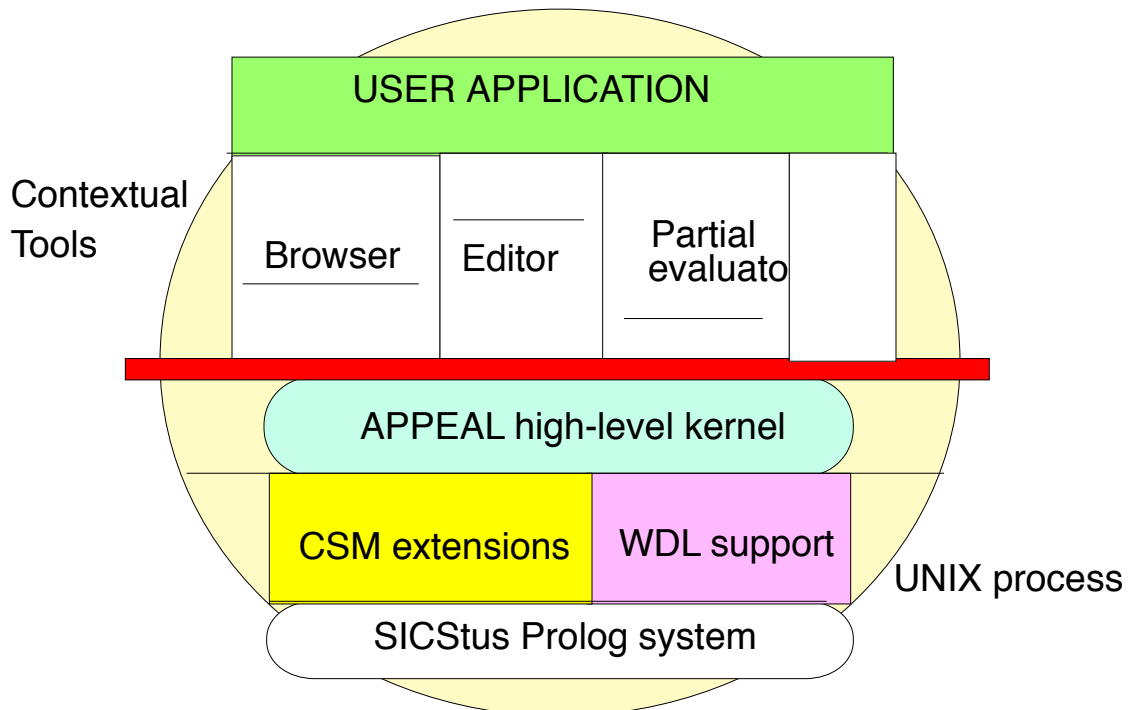
a UNIX process devoted to high-level control

exploiting the **APPEAL** logic programming system

World

an abstraction supporting shared knowledge and agent interaction, hiding implementation details

APPEAL Architecture



SICStus-Prolog layer enhanced with

- support for declarative X-window programming (described at PAP-93)
- support for contextual logic programming

BOTH NON-INTRUSIVE

A set of open, extendible tools

Contextual LP GENESIS

- Basic idea (D.Miller 1986):

logic implication as a tool for modularity.

$T \supset G$ assume T and prove G from P U T

- Incapsulation and scope captured

$(T \supset G1)$ and $G2 :$ T clauses are hidden during the evaluation of $G2$

- Futher steps:

A.Porto, L.Monteiro (89)

non monotonic extension (contexts)

- University of Bologna:
- (E.Denti,E.Lamma, P.Mello, A.Natali, A.Omicini)
- dynamic scoping
 - implementation (WAM-based or not)
 - contexts as first-class objects

Contexts

A context allows several interpretation:

- a binding environment
(with lexical and dynamic scoping)
- a (unmodifiable) object (prototype)
- a class (with inheritance)
- a (modifiable) instance
- a (structured) logic theory
- a current line of reasoning
(Gabbay and Reily 1984)

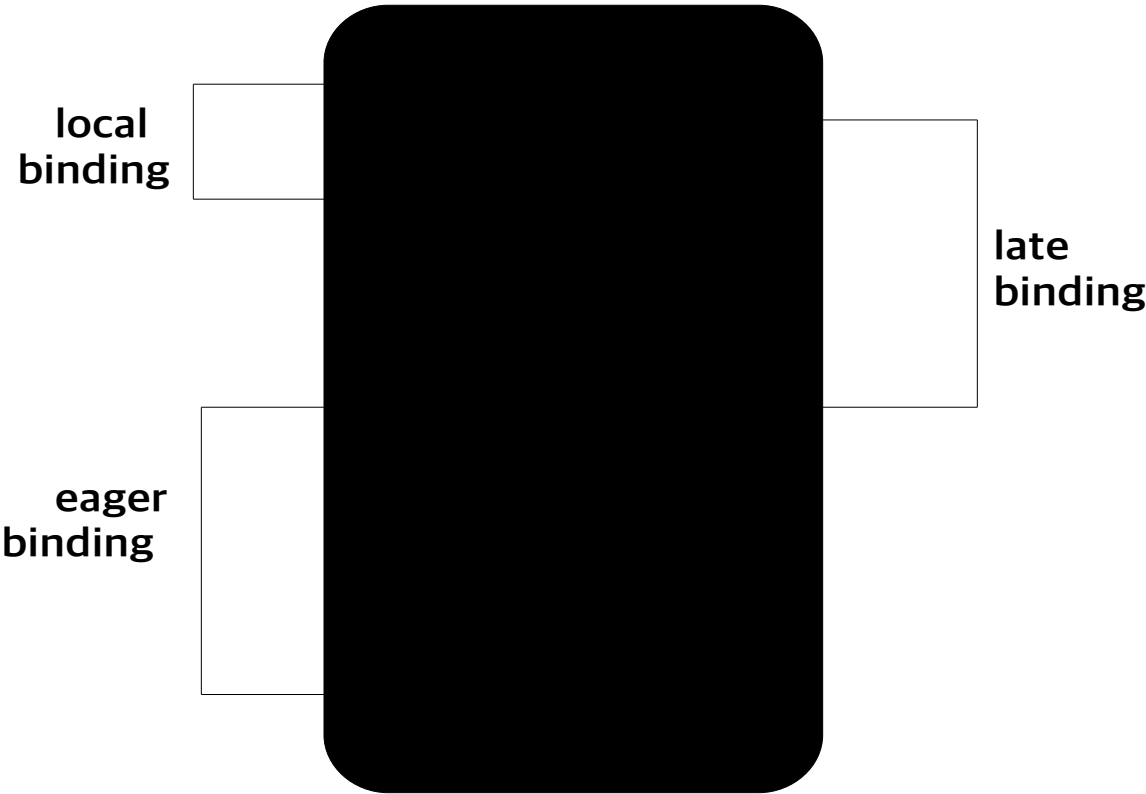
CONTEXTUAL PROGRAMMING IN APPEAL

Unit : a named collection of clauses

- the basic program building block
- an open world
- which can be statically or dynamically composed

Context : a named ordered set of units

u0 >> u1 >> u2 >> Goal



CSM OPERATORS

Constructor

unit/1,
createContext/2, createContext/3

Selectors

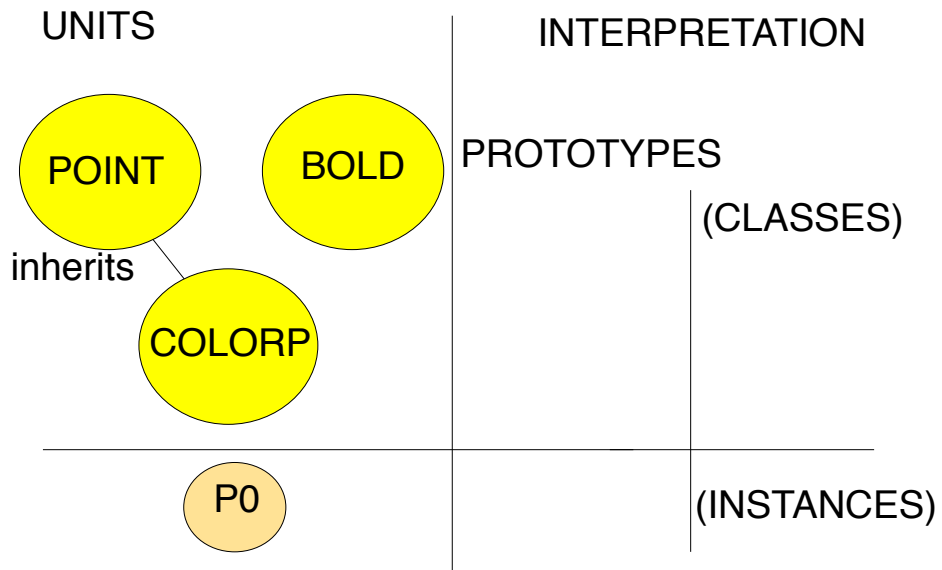
topUnit/2, firstSubcontext/2, unitOf/2, subcontextOf/2,
unitList/2

Static configuration

inherits/2
(visible)/1, (hidden)/1, (extended)/1

State

currentContext/1, currentBindcontext/1
currentSelf/1, currentClass/1, currentSuper/1
whichContext/3, unitPred/2, ctxPred/2,
unit/1, context/1,



```
:- unit point.
size(1).
pos(X,Y):- xelf posxy(X,Y).
draw:-      pos(X,Y), xelf size(Sz),
              drawPoint(X,Y,Sz).
```

```
:- unit colorp inherits point.
color(red).
draw:-      color(C),
              currentColor(X),
              setColor(C),xuper draw,
              setColor(X).
```

```
:- unit bold.
size(20).
```

```
:- unit p0.
posxy(0,0).
```

```
?-      colorp >> p0 >> draw.
```

```
?-      createContext([p0,bold | colorp],p1).
?-      p1<-draw.
```

CRITICAL POINTS

- Classes and dynamic instance creation
can be reduced to *context* and *unit creation*
- Object state
KEY POINT: make a clear distinction between
state *configuration* and *modification*

The CSM approach:

Objects are dynamically created *extension-theories*

Each extension-theory is made of a *fixed, shared* part (the class) and a specific, dynamically configurable part (the object-state knowledge unit)

The object-state knowledge is a set of ground clauses, obtained by *abduction*, under program *constraints*

STATE CONFIGURATION IN CARA

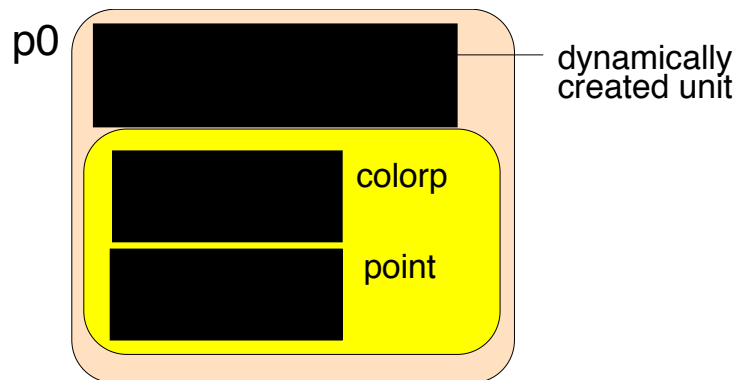
memoization, in ground clause form, of the results of the (deductive) computational process involved by a *constructor* predicate

```
cons :- g1,...,gn  s1,...,sk
```

```
:- unit point.
point(X,Y) :- xelf posxy(X,Y), !.
point(X,Y) :- integer(X),integer(Y)
               posx(X,Y).
...
```

```
:- unit colorp inherits point.
colorp(C,X,Y):- point(X,Y),xelf mycolour(C).
colorp(C,X,Y):- point(X,Y)  mycolour(C).
...
```

```
?-      p0 <== colorp( red,0,0 ).
```



```
?-      p0 <- colorp( C,X,Y ).
C=red, X=0, Y=0,
```

STATE MODIFICATION IN CARA

Temporary

a backtrackable *extension* of the current object-state knowledge unit.

```
:- unit point.
```

```
...
move(DX,DY):-  pos(CX,CY),
                X is CX+DX, Y is CY+DY
                posx(X),posy(Y).
...
```

```
?-      p0 <== colorp( red,0,0 ).      {creates p0}
```

```
?-      p0 <- draw,                      {draw p0 at 0,0}
        p0 <- point(0,0),                {succeeds}
        p0 <- move(10,10),               {moves p0}
        p0 <- draw,                      {draw p0 at 10,10}
        p0 <- point(0,0),                {true(in the past)}
        p0 <- point(X,_),X=0.            {fails}
```

```
no
```

```
?-      p0 <- pos(X,Y).
X=0, Y=0.
```

STATE MODIFICATION IN CARA

Permanent

imperative modification of the object-state knowledge unit.

```
:- unit point.
```

```
...
```

```
domove(DX,DY):-      -posxy(CX,CY),  
                      X is CX+DX, Y is CY+DY,  
                      +posxy(X,Y).
```

```
...
```

```
?-      p0 <== colorp( red,0,0 ).      {creates p0}
```

```
?-      p0 <- domove(10,10),           {moves p0}  
      p0 <- draw,                      {draw p0 at 10,10}  
      p0 <- point(0,0).                 {fails}
```

```
?-      p0 <- point(X,Y).  
X=10, Y=10.
```

THE IMPACT OF CONTEXTUAL LP

- Contexts provide a bridge between static program organizations based on class and inheritance and evolutive software construction

EVOLUTIVE SOFTWARE can be intended as:

- the conventional evolution (defined and controlled by the user) which takes place in a programming environment
- the modification of the application software architecture by the application system itself

IN ANY CASE: a constraint-based process

- LP can help in delegating to the system constraint verification/satisfaction

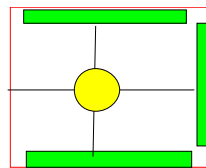
AN EXAMPLE: EVOLUTIVE AGENTS

Agents with different skills and assumptions can be statically defined and dynamically composed according to events or needs.

Basic Agent

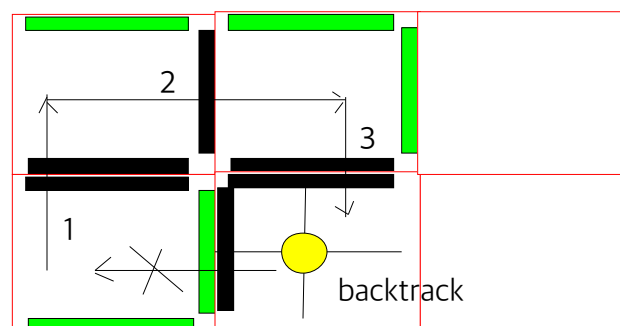
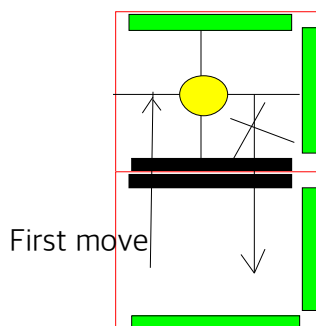
- reacts to the first input it receives
- no reasoning activity
- *open* with respect to

<code>getInput(X)</code> <code>move(Dir)</code>	<code>get a data from world</code> <code>move in direction Dir</code>
--	--



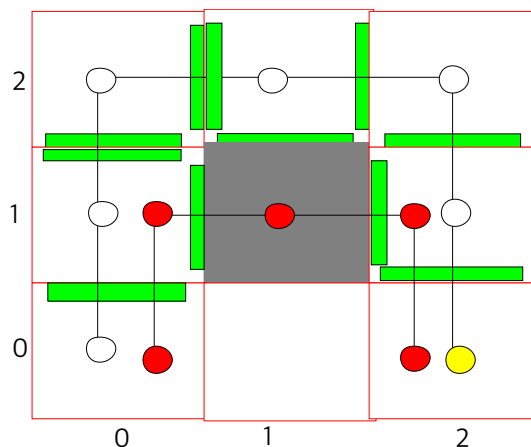
Clever agent

- meta-control: performs useful actions only
- subsuming basic behaviour
 redefines move and getinput
- revoking moves on failure



Smart agent

- situated reasoning:
 - all knowledge acquired is available
 - reasoning on incomplete knowledge
 - units can be used as plans
 - contextual configuration as current line of reasoning



WORLD knowledge

```
message( cell(0,0,n), 1
          cell(0,1,n), 2
          cell(0,1,e), 3
          cell(0,2,s), 4
          cell(0,2,e), 5
          cell(1,2,w), 6
          cell(1,2,e), 7
          cell(1,2,s), 8
          cell(2,2,s), 9
          cell(2,1,w), 10
          cell(2,1,s), 11
        )
```

Smart agent strategy

```

react(__, __, pos(N1,N2), pos(N1,N2)) :- !.

react(_, RevP, pos(N1,N2), pos(D1,D2)) :-
(D1 > N1 ->
    toeast >:> action(RevP,pos(D1,D2));
 D1 < N1 ->
    towest >:> action(RevP,pos(D1,D2));
 D1 = N1 ->
    (D2 > N2 ->
        tonorth >:> action(RevP,pos(D1,D2));
        tosouth >:> action(RevP,pos(D1,D2))
    )
).

```

- When the system is extended with `toeast` unit, a plan is set where *east* is the first direction tried, followed by *south* and *north*. *West* is the last choice .
- The contextual configuration of the agent set by the cactus extensions `T >:> G` can be interpreted as its current *line of reasoning*.

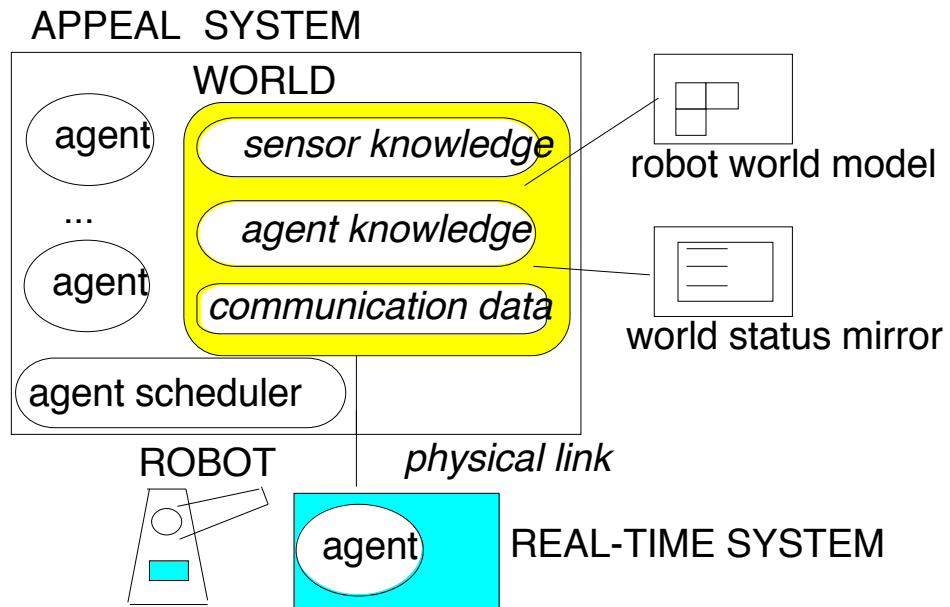
```
:- unit basicAgent.
agent(Path,pos(N1,N2)):-
    agent([],pos(N1,N2),Path).
agent(RevP,pos(N1,N2),Path):-
    curPos(N1,N2),!, % goal achieved
    reverse(RevP,Path).
agent(RevP,Dest,_):-
    curPos(N1,N2),
    #getInfo(pos(N1,N2),Dir),
    #react(Dir,RevP,pos(N1,N2),Dest).
getInfo(pos(N1,N2),Dir):-
    activateSensors(pos(N1,N2)),
    #getInput(cell(N1,N2,Dir).
react(Dir,RevP,_,Dest):-
    #action(Dir,RevP,Dest).
action(Dir,RevP,Dest):-
    #move(Dir),
    updatePos(Dir),
    agent([Dir|RevP],Dest,_).

getInput(X):-world <- rd(X).
move(Dir):-
    myname(P),
    world<-out(move(P,Dir)),
    world<-in(endmove(P,Dir)).
```

Robot Programming

- support for different architectures
- support for prototyping techniques
- object-oriented modelling capabilities
- integration of different technologies
- tools for symbolic computations as well as for event-driven computations
- support for parallelism (DAI)

THE WORLD ABSTRACTION



World: an APPEAL object

including shared knowledge of an agent society

working as an interaction device in the Linda style

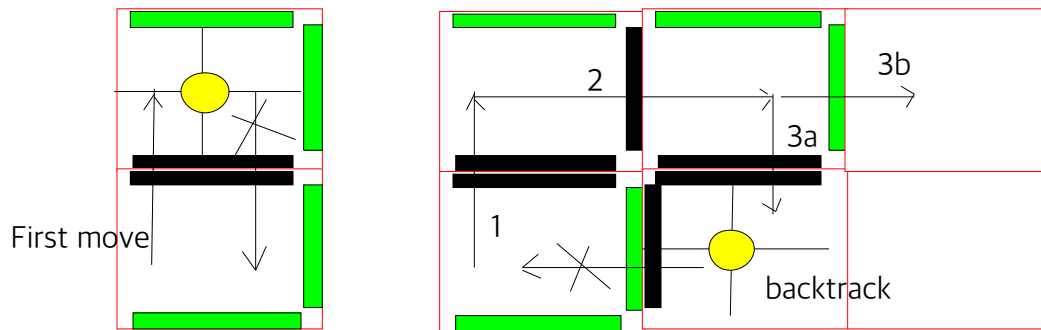
- time uncoupling
- communication orthogonality

with peculiar features:

- associative access to message information (unification vs. pattern matching)
- integration between event-handling programming models and reasoning
- full extendibility by users

Extending communication mechanisms

Clever agent



Event handling has to be integrated with reasoning.
When new data arrive from sensors:

- a waiting agent must be reactivated
- the agent must consume data only "logically"
- the agent may backtrack on failure and try other messages
- when no (other) sensor-data are available, the agent must fail

a new primitive: `rdCWA(Msg)`

```
:- unit newWorldClass inherits worldClass.

newWorldClass :-
    newName(C1), C1 <=< counter(0) | clock(C1).

updateTime(T) :-
    world <- clock(C), C <- tick(T).

currentTime(T) :-
    world <- clock(C), C <- counter(T).

out(Msg) :-
    updateTime(CurT),
    +message(Msg,CurT),
    scheduler <- reactivate(Msg).

in(Msg) :- ...
    % rewritten according to new Msg format

rd(Msg) :- ...
    % rewritten according to new Msg format
```

```

rdCWA(Msg) :-
    myname(P),
    freevars(Msg,MsgTemplate),
    #lastAccess(MsgTemplate,P,TOld),!,
    getMsg(Msg,P,TOld).

getMsg(Msg,P,TOld) :-
    % if there are messages
    existsNewMsg(Msg,TOld),!,
    % then take them
    currentTime(TLimit),
    takeMsg(Msg,P,TOld,TLimit).

getMsg(Msg,P,TOld):-
    % otherwise suspend the agent
    scheduler <- suspend(Msg,P),
    % and retry when reactivated
    getMsg(Msg,P,TOld).

takeMsg(Msg,P,TOld,TLimit) :-
    % logically consume all messages (in backtracking)
    % between TOld and TLimit
    #message(Msg,TNew),
    TNew > TOld, TNew =< TLimit,
    % and update the agent last access time
    updateLastAccess(P,Msg,TNew).

takeMsg(Msg,P,_,TLimit):-
    % look for further messages beyond TLimit
    existsNewMsg(Msg,TLimit),
    % if there are messages, then consume them
    currentTime(TCur),
    takeMsg(Msg,P,TLimit,TCur).
    % otherwise fail.

updateLastAccess(P,Msg,TNew) :-
    freeVars(Msg,MsgTemplate)
    | lastAccess(MsgTemplate,P,TNew).

existsNewMsg(Msg,TOld) :-
    copy_term(Msg,MsgTemplate),
    #message(MsgTemplate,TNew),
    TNew > TOld.

```

SUPPORT FOR TESTING AND DEBUGGING:

Contextual-based system programming can be exploited to *extend kernels* rather than modify them.

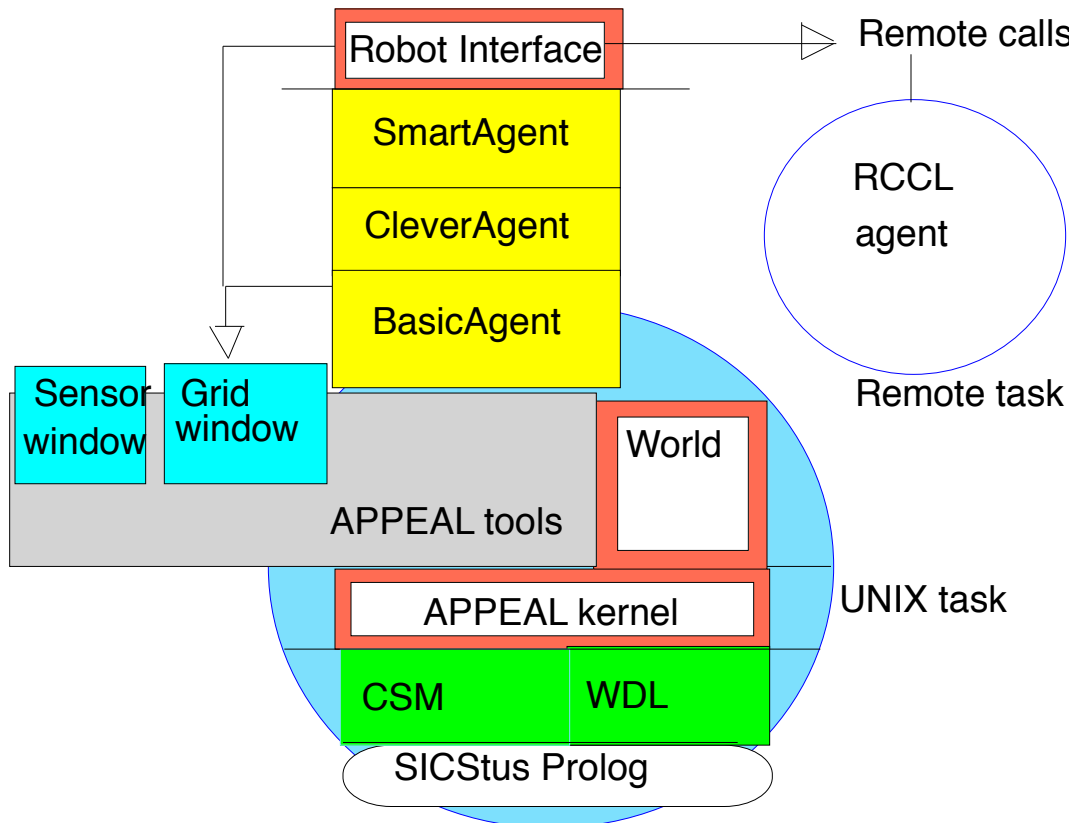
First phase:

a local arm simulator is used

Second phase:

any call to local "move" primitives is redirected to an external process.

what is normally a kernel modification can be introduced as a **KERNEL EXTENSION**



CONCLUSIONS

APPEAL and CARA exploit and support the power of incremental software construction

- SICStus Prolog
- X-window declarative programming support
- Contexts
- Declarative object-oriented programming
- Imperative object-oriented programming
- High-level, integrated, extendible support to agent interaction

Each extension has its application niche.

With respect to conventional languages and tools, peculiar features are provided.

In the robotic field, both symbolic and reactive systems can be expressed

Next step (end of 1994): working with the real system

CONCLUSIONS

(SICStus) Prolog	LP in use
modules	useful, but not enough
event handling	X-window declarative interface
contexts	rooted in a declarative model nice, but not enough
named contexts (CSM)	persistency assured.
	Prototyping and delegation
contexts + local assignment + event-handling	conventional objects communication units
contexts + extendible-only units	evolving, backtrackable objects
contexts + abduction + constraints	declarative objects

Each feature has its application niche.

They should make easier the transition from conventional software design and development to *logic* programming.

A STRUCTURED LOGIC PROGRAMMING APPROACH TO ROBOT PROGRAMMING

ANTONIO NATALI

ENRICO DENTI

ANDREA OMICINI

FRANCESCO ZANICHELLI

CNR Project on New Programming Languages
CNR Project on Robotics

Second Conference on the Practical Application of Prolog

PAP '94

London (UK), April 29, 1994

THE CONTROL ISSUE

Two basic needs:

- asynchronous event handling (e.g. X-window events)
 - a basic feature of APPEAL
- support for agent society metaphor

BACKGROUND

Parallel interpretation of clauses (STREAMS)

- not Prolog-compatible
- no explicit control on process scheduling
- low-level, point-to-point communication model

Communicating Prolog Units (CPU)

- communication through Prolog databases
- in the Linda style
 - time uncoupling
 - communication orthogonality
- shared knowledge in clausal form
- full control over process scheduling
- simple integration in the contextual model

COMMUNICATING PROLOG UNITS

```
activate(PID,Goal)
```

CPU processes do not share logic variables. They interact through *communicating units*.

```
CU <- m(X)           a query (CU as a closed world)
```

```
CU <- rd(m(X))      a suspensive query.
```

```
CU <- in(m(X))      like rd, but changes CU
```

```
CU <- out(m(a))     extends CU with message m(a)
```

```
:- unit scheduler.
```

```
suspend(Msg):- #curr_process(P),
               +waiting(Msg,P),
               -waiting(processor,S),!,
               transfer(S).
```

```
suspend(Msg):- #idle_process(P),
               transfer(P).
```

```
reactivate(Msg):- -waiting(Msg,P),!,
                  +waiting(processor,P).
```

```
reactivate(Msg).
```

State knowledge

curr_process(P)	P is <i>running</i>
waiting(processor,P)	P is <i>ready to run</i>
waiting(Msg,P)	P <i>waits a message</i>

```
:- unit worldClass.
```

```
in(Msg) :-  
  %consume the first available msg  
  (   -Msg,! ;  
  % otherwise suspend the process  
      scheduler <- suspend(Msg),  
  % and retry when reactivated  
      in(Msg)  
  ).
```

```
out(Msg):-  
  % make the message available  
      +Msg,  
  % and reactivate a waiting process  
      scheduler <- reactivate(Msg).
```

```
rd(Msg):-  
  % read the first available message  
  (   #Msg,! ;  
  % otherwise suspend the process  
      scheduler <- suspend(Msg),  
  % and retry when reactivated  
      rd(Msg)  
  ).
```

```
emit(Msg):-  
  % store a backtrackable message  
      | Msg.
```