Agent Coordination and Control through Logic Theories

Andrea Omicini, Enrico Denti, Antonio Natali

Dipartimento di Elettronica, Informatica e Sistemistica
Università degli Studi di Bologna - Italy
email: {edenti, anatali, aomicini}@deis.unibo.it

Abstract. This work describes an agent interaction model (*ACLT*, Agent Communicating through Logic Theories) rooted in the concept of logic theory. *ACLT* agents and their behaviour are conceived as inferential as well as procedural activities within a multiple theory space. The communication unit (CU) abstraction is exploited, subsuming traditional communication models (both shared memory and message passing) based on explicit and extensional knowledge, while allowing agents to exploit partial/incomplete knowledge through deduction. Agent synchronization is reconducted to the concept of theory evolution, by allowing agents to wait for theory modification until facts can be deduced from a CU. Agent cooperation/competition is re-interpreted in terms of knowledge generation/consumption. A coherent notion of logic consequence in a time-dependent environment is proposed. As a result, the traditional dichotomy between reactive and symbolic systems is here exploited as a feature rather than a problem, leading to an integration of behavioural and planning-based approaches.

1 Introduction

The abstract model of interaction between agents, as well as the balance between high-level agent coordination/management and low-level communication/control, are among the most critical issues in the design and implementation of multiagent systems.

The (high-level) notions of agent and agent coordination cannot be simply reconducted to the (low-level) concepts of process, and process communication/synchronization. Metaphors for communication abstracted from the physical support by which the interaction takes place (like monitors, derived from the idea of using shared memory as a communication device, or message passing, inspired by communication via connection lines) are well suited in the development of machine-level systems, such as operating systems: however, higher-level communication models and abstractions are required to capture the idea of a general-purpose, hardware independent multiagent architecture.

In particular, it is often the case that a multiagent system is designed (or described, once built) as a knowledge-based system where agents interact not simply by exchanging raw information (such as messages or signals), but by sharing and exchanging knowledge. This mainly occurs when the information exchanged by agents can be interpreted as a (partial) model of the world (the application domain), so

that agents can exploit that knowledge in order to deduce new facts that are not explicitly generated, exchanged or stored.

In order to promote a knowledge-based approach to agent interaction, new communication abstractions have been proposed, such as the concept of blackboard [1] or the idea of a tuple space as a (generative) communication device [2]. The main benefits of such abstractions are related to the following properties:

- agent interaction can be expressed in a very simple way
- agents can be heterogeneous (the blackboard or the tuple-space acts as a coordination device)
- agent do not need to know each other (agent name independence)
- time uncoupling

However, what these models lack is an effective way to exploit (shared) information as true knowledge, since they provide no means to infer new information from it. This would be useful not only for multiagent systems based on explicit world-modelling and symbolic reasoning, but also for reactive systems, since reactions can take place as a consequence of an inference.

The aim of this work is to show the benefits of endowing a generative communication model with inferential capabilities for designing multiagent systems. The $\mathcal{ACLT}$ (Agents Communicating through Logic Theories) model proposed in this paper is a first-order logic-based approach, rooted in the CPU programming model described in [3]. Its key idea is to take logic theory as an abstract data type which clients can perform three kinds of operations on:

- Linda-like communication operations, based on side-effect,
- logic inferences, based on don't know non-determinism, and
- hybrid operations.

The $\mathcal{ACLT}$ model allows multiagent system designers to obtain a good balance between high-level agent coordination and management and (the cost of) low-level communication/control. In addition, this approach promotes hybrid architectures in which pure reactive behaviours can be naturally integrated with high-level symbolic activities, based on reasoning and planning. In the work we will give examples of these capabilities, by discussing in particular some problems originating from the field of autonomous robot control.

This paper is structured as follows. Section 2 introduces the basics of the $\mathcal{ACLT}$ programming model, as implemented in a multiple theory logic language based on Prolog. Section 3 discusses the implications involved with the coexistence of time-related changes and logic inferences in the $\mathcal{ACLT}$ model, by presenting some $\mathcal{ACLT}$ hybrid primitives. After a short sketch of the actual implementation, given in Section 4, Section 5 is devoted to final remarks and conclusions.

2 The $\mathcal{ACLT}$ model

2.1 Basic communication primitives

The $\mathcal{ACLT}$ model for multiagent systems is rooted in the CPU model [3], and is founded on the notion of *logic theory* as a coordination device.

An $\mathcal{ACLT}$ agent is a process generated by an $\mathcal{ACLT}$ sequential logic program. However, $\mathcal{ACLT}$ allows any process able to accomplish the basic communication protocol ($\mathcal{ACLT}$ non-backtrackable Linda-like primitives) to interact with the whole multiagent system. A new $\mathcal{ACLT}$ agent is started through the `activate/2` primitive: `activate(Theory, Goal)` generates a new agent as a concurrent logic process trying to derive `Goal` from `Theory`.

A multiplicity of $\mathcal{ACLT}$ agents communicate through a collection of unitary clauses by means of Linda-like operations (*in*, *out*, and *read*). Such primitives work on a clause database used as conventional Linda tuple space, by inserting, removing or accessing *logic tuples* (à la Shared Prolog [4]). Thus, the main features of Linda as a coordination language (time uncoupling, communication orthogonality) and its expressive power can be fully captured in $\mathcal{ACLT}$.

Following [5], $\mathcal{ACLT}$ relaxes original Linda constraint of a unique tuple space, by allowing many communication units to be defined and used independently by a collection of agents. An $\mathcal{ACLT}$ *communication unit* (CU) is a logic theory, denoted by a ground name, to be used as a logic tuple space. The following syntax

```
:- commUnit world.
```

declares that `world` denotes an $\mathcal{ACLT}$ communication unit, which can be subsequently associated to any $\mathcal{ACLT}$ communication primitive. Thus,

```
out(p( t̃ ))@world
```

inserts logic tuple `p( t̃ )`(where `t̃` is a tuple of terms) into CU `world`. Of course, `in` and `read` primitives can be used the same way:

```
in(p( t̃ ))@world
read(p( t̃ ))@world
```

In addition, $\mathcal{ACLT}$ provides the full power of *unification* instead of the pure Linda pattern matching. Thus, an agent calling `read(p( t̃ ))@world` is suspended until a logic tuple `p( t̃ ′)` *unifying* with `p( t̃ )` can be found in theory `world`. Then, agent is resumed after unification of `p( t̃ )` and `p( t̃ ′)`. Analogous considerations can be repeated for `in` primitive, except for the removal of `p( t̃ ′)` from `world`, occurring between unification and agent resumption.

2.2 Communication units as logic theories

However, the main feature of the logic-based $\mathcal{ACLT}$ model comes from fully exploiting the twofold interpretation of the CU abstraction. Information shared by agents through CUs can be interpreted as knowledge about the world. In this view, agents do not simply exchange messages through the tuple space, but behave as knowledge sources (such as in a blackboard-based system), each one giving a partial

description of the world. Thus, an $\mathcal{ACLT}$ agent can exploit a CU in two distinct ways, (*i*) as a communication device, and (*ii*) as a logic theory describing the application domain.

The main consequence of interpreting the tuple space information as knowledge in a logic framework, is the chance to perform logic inference operations based on CU axioms. $\mathcal{ACLT}$ logic-based model provides primitives such as

```
demo(p( t̃ ))@world
```

whose intended semantics follows the typical inference mechanism of a logic language based on don't know nondeterminism, like Prolog. In fact, `demo(p( t̃ ))@world` succeeds iff `p( t̃ )` logically follows from logic theory `world`.

Given that CUs contain only unitary clauses, it might be observed that a very limited notion of logical consequence is used here. However, since the problems of non-deterministic exploration of a knowledge base are basically unaffected by this restriction, this seems not to represent an issue in our framework.

The following simple example shows how Linda-like "standard" primitives and logic-based primitives like *demo* can fruitfully coexist in a multiagent framework.

```
:- unit speedyMouse.

startAt(FX,FY) :-
  read(cheeseAt(TX,TY))@grid,
  in(freeCell(FX,FY))@grid,
  speedyMove(FX,FY,TX,TY,[]),
  out(cheeseFoundAt(TX,TY))@grid.

speedyMove(FX,FY,TX,TY,RPath) :-
  activateDirSensors(FX,FY),
  demoWaitLast(door(FX,FY,Dir))@grid,
  \+ loop([Dir|RPath]),
  nextCell(Dir,FX,FY,X,Y),
  in(freeCell(X,Y))@grid,
  moveTo(X,Y),
  out(freeCell(FX,FY))@grid
  ( X = TX, Y = TY -> true;
    speedyMove(X,Y,TX,TY,[Dir|RPath]) ).

loop([]) :- !, fail.
loop([_]) :- !, fail.
loop([Dir|Path]) :- loop(Path).
loop(Path) :- relMove(Path,(0,0),(0,0)).

relMove([],(X,Y),(X,Y)).
relMove( [Dir|Path], (X,Y), (TX,TY) ) :-
  ( Dir = north -> NY = Y+1, NX = X;
    Dir = south -> NY = Y-1, NX = X;
    Dir = east  -> NX = X+1, NY = Y;
    Dir = west  -> NX = X-1, NY = Y ),
  relMove(Path,(NX,NY),(TX,TY)).

nextCell(west,X,Y,NX,Y) :- NX is X - 1.
nextCell(east,X,Y,NX,Y) :- NX is X + 1.
nextCell(south,X,Y,X,NY) :- NY is Y - 1.
nextCell(north,X,Y,X,NY) :- NY is Y + 1.

...
```

```
:- unit lazyMouse.

startAt(FX,FY,M,N) :-
  freeCells(M,N),
  activateCheeseSensor,
  read(cheeseFoundAt(X,Y))@grid,
  smartMove(FX,FY,TX,TY,0).

smartMove(FX,FY,TX,TY,Dev) :-
  plan(FX,FY,TX,TY,Dev,Path,[])
  -> moveAlong(Path)
  ;  NDev is Dev + 1,
     smartMove(FX,FY,TX,TY,NDev).

plan(X,Y,X,Y,_,Path,RPath) :-
  reverse(RPath,Path).
plan(FX,FY,TX,TY,Dev,Path,RPath) :-
  getCellDoors(FX,FY,D),
  ( noDev(D,FX,FY,TX,TY,X,Y)
  -> plan(X,Y,TX,TY,Dev,Path,[D|RPath])
  ;  Dev > 0, NDev is Dev - 1
     plan(X,Y,TX,TY,NDev,Path,[D|RPath]) ).

noDev(D,X,Y,TX,TY,NX,NY) :-
  D = west, X > TX -> NX is X - 1, NY = Y;
  D = east, X < TX -> NX is X + 1, NY = Y;
  D = south,Y > TY -> NY is Y - 1, NX = X;
  D = north,Y < TY -> NY is Y + 1, NX = X.

getCellDoors(X,Y,D) :-
  demo(door(X,Y,D))@grid;
  ( D = west  -> NX is X-1, NY=Y, OD = east;
    D = east  -> NX is X+1, NY=Y, OD = west;
    D = south -> NY is Y-1, NX=X, OD = north;
    D = north -> NY is Y+1, NX=X, OD = south ),
  demo(door(NX,NY,OD))@grid.

...
```

Fig. 1. Speedy (a) and lazy (b) mouse code

2.3 The Mouse Agents Example

Two sorts of mouse agents (speedy mice and lazy mice) live in a 2-dimensional space, represented by a $m \times n$ grid, whose cells (denoted by a pair of coordinates) are separated by either *doors* or *walls*. Mouse agent's aim is to reach a piece of cheese, located at a given cell of the grid, by moving across the grid. Mice can pass doors, but obviously cannot cross walls.

Each speedy mouse features four sensors, one for each basic direction (north, south, etc.), to tell doors from walls, assuming that there is always at least one door for any cell. Sensor activation is under the explicit agent control through the primitive `activateDirSensors(X,Y)`, which results in storing one to four facts of the form `door(X,Y,Dir)` in the CU `grid`, through an `out` primitive triggered by physical (low-level) sensors. Each `door(X,Y,Dir)` fact denotes the presence of a door allowing agents to move from cell `(X,Y)` in direction `Dir`. Instead, no explicit information is stored about wall positions.

Speedy mouse initially waits for information about cheese location coming from lazy mouse. Thus, it is initially suspended on a `read(cheeseAt(X,Y))` operation on the CU `grid`. When a fact such as `cheeseAt/2` is inserted in `grid` by lazy mouse's cheese sensors, speedy mouse can start its exploration.

Speedy mice cannot occupy the same cell at the same time. These agents wait for a fact `freeCell(X,Y)` (through a `in`) before moving to cell `(X,Y)`, and insert the same fact when stepping out from there. Whenever a new (i.e., never previously visited) cell is entered, a speedy mouse agent activates direction sensors, then chooses where to go according to a simple non-loop strategy. Should two or more directions be possible, one is chosen nondeterministically (while the others are possibly reconsidered on backtracking). When cheese is finally reached, speedy mouse outputs a fact of the form `cheeseFoundAt(X,Y)`, thus triggering lazy mouse agents.

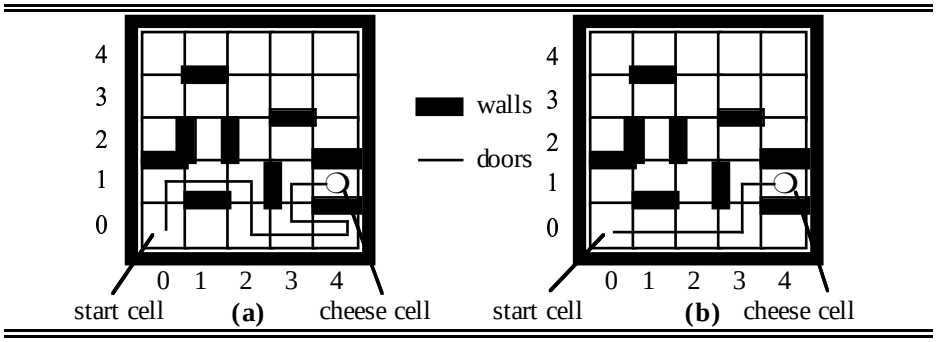

Fig. 2. Speedy (a) and lazy (b) mouse paths in a 5x5 grid

Figures 2 refer to the simple case of two agents (one speedy and one lazy mouse), both starting at the (0,0) cell of a 5×5 grid, and looking for cheese located at (4,1). The following goal leads to the activation of such a 2-agent system.

```
:- commUnit(grid),
   activate(lazyMouse, startAt(0,0,5,5)),
   activate(speedyMouse, startAt(0,0)).
```

It is easy to see how more mice could be started at the same time, even though the trivial protocol used here for free cell detection would be obviously not able to prevent even the simplest case of deadlock.

During its exploration of the grid space, speedy mouse works as a source for information about grid configuration. Such knowledge remains available in the `grid` CU. Figure 3 shows `grid` status after speedy mouse's exploration of Figure 2(a).

```
:-  commUnit grid.
freeCell(0,2).   freeCell(3,4).    freeCell(0,0).    freeCell(2,0).
freeCell(0,3).   freeCell(3,5).    door(0,1,south).  door(3,0,east).
freeCell(0,4).   freeCell(4,2).    door(0,1,east).   door(3,0,north).
freeCell(0,5).   freeCell(4,3).    freeCell(0,1).    door(3,0,west).
freeCell(1,0).   freeCell(4,4).    door(1,1,east).   door(4,0,west).
freeCell(1,2).   freeCell(4,5).    door(1,1,north).  freeCell(4,0).
freeCell(1,3).   freeCell(5,0).    door(1,1,west).   freeCell(3,0).
freeCell(1,4).   freeCell(5,1).    freeCell(1,1).    door(3,1,south).
freeCell(1,5).   freeCell(5,2).    door(2,1,south).  door(3,1,east).
freeCell(2,2).   freeCell(5,3).    door(2,1,west).   door(3,1,north).
freeCell(2,3).   freeCell(5,4).    door(2,1,north).  freeCell(3,1).
freeCell(2,4).   freeCell(5,5).    freeCell(2,1).    cheeseFoundAt(4,1).
freeCell(2,5).   cheeseAt(4,1).    door(2,0,east).
freeCell(3,2).   door(0,0,north).  door(2,0,west).
freeCell(3,3).   door(0,0,east).   door(2,0,north).
```

Fig. 3. CU `grid` configuration after speedy mouse exploration

Unlike speedy mice, lazy mouse has no direction sensors. Instead, it is provided with a cheese sensor, which gets it to detect cheese position. In fact, a `cheeseAt(X,Y)` fact is initially inserted in `grid` as a result of `activateCheeseSensor` lazy mouse's invocation.

When cheese is reached, speedy mouse awakens lazy mouse, previously suspended on a `read(cheeseFoundAt(X,Y))@grid`. With respect to speedy mouse, lazy mouse adopts a more sophisticated, intelligent strategy, since it *reasons* over already-available knowledge to identify the best path. In the case shown in Figure 2, lazy mouse recognizes that a better path exists with respect to the one used by speedy mouse, and moves straight to the cheese position (indicated by a flag).

This example clearly points out (*i*) how a CU can be used at the same time both as a synchronization device and as a knowledge repository, as well as (*ii*) how these two interpretations can coexist fruitfully. In particular, this can be seen from sensor information about grid doors: generated by a low-level agent, this information is used first in a reactive fashion by speedy mouse, which waits for sensor input before moving, then as a knowledge base for a planning activity by lazy mouse, inferring the best path to cheese. Moreover, the example highlights how the abstractions defined can be exploited to lift up at the symbolic level some typical, low-level reactive behaviours.

3 Logic consequence and time-related changes

3.1 $\mathcal{ACLT}$ primitives for logic inference

The main problem of using a logic-based language as a coordination language lays in side-effect nature of the communication primitives. While the interpretation of a clause database as communication device is bound to a vision of dynamic, evolving information, the logic theory reading is instead founded over a notion of platonic, universal truth. The main issue of this paper is then how to make the two readings coexist in the same conceptual framework, by pointing out at the same time some benefits of this integration.

The Mouse Agents Example intentionally hides some of the problems connected to the twofold interpretation. Initially, nothing is known about grid doors and walls. Then, information about grid structure grows with speedy mice's exploration, and stops growing when such a task is brought to end. As a result, when lazy mice start moving, CU `grid` contains a (possibly partial) description of the grid space, which will remain unchanged if no other speedy mouse is started. In this case, such a knowledge can be used for reasoning activities with no particular caution: nothing changes in the logic theory during lazy mouse inference operations.

The problem would arise for instance in case the lazy mouse would have started its reasoning while some speedy mouse is still moving. In that case, sequential exploration of a set of axioms under evolution would pose a problem, due its cost in terms of *computational time*: which knowledge have to be considered for the construction of a logic proof in an evolving knowledge space?

In this connection, it is useful to discuss the relation between time-related (such as communication primitives based on side-effect) and time-independent operations (such as logic inference) by defining the class of the *demo* primitives provided by $\mathcal{ACLT}$.

Till now, we only said that `demo(p(`$\tilde{t}$`))@world` succeeds if `p(`$\tilde{t}$`)` logically follows from `world`. However, this semantic specification is not satisfactory at all, given that `world` is not a statically defined collection of axioms, but rather a clause database which evolves during the computation. Sequential exploration of an axiom space has a cost in terms of computational time, and the knowledge related to `p` predicate may grow or shrink during the exploration.

However, the notion of logic consequence lays on the assumption of a *fixed* axiom set defining the space of the theorems. Thus, in order to define a coherent notion of logic consequence in a time-dependent environment, we choose to set an instant when to freeze time, so as to perform time-independent activities. As a result, each *demo* operation has to be associated to a given *snapshot* of the involved CU, where to perform safely sequential proofs.

Thus, different semantics for *demo* can be defined according to different definitions of when a communication unit snapshot has to be taken to be associated conceptually to the *demo* operation. Alternatively, different primitives of the *demo* class can be defined by taking the snapshot at different moments.

In particular, $\mathcal{ACLT}$ provides four basic *demo* primitives: `demo`, `demoWait`, `demoLast`, and `demoWaitLast`. If the operation is performed at *t*, first served at *t'*, and

(possibly) fails at *t"*, then the snapshots associated to the primitives are conceptually taken at *t*, *t'*, *t"*, and (again) *t"*, respectively. In detail, the intuitive semantics of the four primitives can be given conceptually as follows.

When a `demo(p( t̃))@world` operation is performed by an $\mathcal{ACLT}$ agent, a `p( t̃)`-snapshot of `world` (that is, those axioms of `world` unifying with `p( t̃)`) is immediately (at *t*) taken, and bound to that primitive activation, which is immediately served (*t* = *t'*). If this snapshot is empty, then the call immediately fails (*t'* = *t"*). Instead, if some facts unifying with `p( t̃)` exist in `world`, then a logic derivation is performed, using standard Prolog computational rule. If no branch of the derivation succeeds, the computation finally fails (*t'* < *t"*, where Δ*t* = *t"* - *t'* is the non-null computational cost of the exploration of the `p` axioms).

In the Mouse Agents example, for instance, lazy mouse performs several `demo(door(X,Y,Dir))` operations (see Fig. 1) in order to derive the best path from information about grid doors provided by speedy mouse. For instance, when the first call `demo(door(0,0,Dir))@grid` is performed by lazy mouse as an indirect result of `smartMove` in order to step out from the start cell, the snapshot of `grid` associated to that *demo* operation consists of the two facts {`door(0,0,north), door(0,0,east)`} (see Figure 2). Any further modification to `grid` theory (for instance, by a third mouse agent) has no effect on `door(0,0,Dir)` demonstration.

Instead, when a `demoWait(p( t̃))@world` operation is performed, the calling agent is suspended until some suitable knowledge is found in. The `p( t̃)`-snapshot of `world` is then taken when the operation is first served (at *t'*). No suspension takes place iff at *t* some axioms unifying with `p( t̃)` is already available: in that case, `demo` and `demoWait` semantics perfectly match (since *t* = *t'*). They differ, instead, when no suitable fact is found in `world` at *t*, so that the snapshot is taken at *t'* > *t*. Since `demoWait` is always associated to a theory containing suitable knowledge, the computational cost of its logic proof is never null, so that *t'* < *t"* always holds.

Instead, `demoLast(p( t̃))@world` has no suspensive semantics (*t* = *t'*). It has actually the same behaviour as `demo(p( t̃))@world` in case no facts unifying with `p( t̃)` can be found at *t* in `world`: it immediately fails (*t'* = *t"*). However, `world` snapshot is not taken when the operation is performed: instead, `demoLast` try to exploit all the suitable knowledge of `world`, including that one generated *after* the operation was first served (after *t* = *t'*). Thus, if some suitable axioms exists at *t* in `world`, they all are taken into account. In addition, since the cost of sequentially exploring them all is not null, further axioms unifying with `p( t̃)` may have been inserted in `world` during the corresponding computation. In that case, this new information too is to be used for logic inference. Such an operation is repeated until all suitable facts in CU `world` have been tried (with failure) and no further facts unifying with `p( t̃)` have been added. Then, and only then, `demoLast(p( t̃))@world` fails. Since failure occurs only when none of the axioms which can be found in `world` at failure time has produced a successful branch, this amounts to say that `p( t̃)`-snapshot of `world` associated to `demoLast` has been taken at *t"*.

As its name suggests, `demoWaitLast` combines the semantics of `demoLast` and `demoWait`. `demoWaitLast(p( t̃))@world` works like `demoWait` when no suitable knowledge is found in `world`: thus, it never fails immediately, and *t"* > *t'* always

holds. However, when `world` contains some facts unifying with `p( t̃ )`, `demoWaitLast` behave exactly like `demoLast`, by trying to exploit any suitable knowledge made available in `world` at any time.

One example of `demoWaitLast` application has been hidden in the Mouse Agents Example. How can speedy mouse ask for doors, keep itself synchronized waiting for different knowledge sources (its direction sensors) making information available, and then exploit it at its best without polling or time-outs? Its suspensive semantics ensures that the speedy mouse agent waits for at least one sensor response when performing `demoWaitLast(door(FX,FY,Dir))@grid`. In addition, its delay in taking the snapshot presumably avoids that only the first `door(FX,FY,Dir)` sensor information arrived is taken into account. In a situation where we have four distinct sensors of the same type, having then very similar (even though not identical) response time, all `door(X,Y,Dir)` facts relative to a cell `(X,Y)` are usually considered by speedy mouse's `demoWaitLast`. Thus, `demoWaitLast` is actually used as an hybrid primitive: it first produces a synchronizing behaviour, since it allows speedy mouse to wait for at least one door to be detected before start moving. But it is then used also for inference activities since it allows backtracking and possibly exploiting alternative choices on failure.

3.2 Knowledge classification

However, even though this approach seems to be effective from an operational viewpoint, it may lead to inconsistencies from a conceptual viewpoint. What happens, in fact, if some agent withdraws from the tuple space one axiom which belongs to a demo snapshot currently under execution? Operationally, no problem arises. On the other hand, it may happen that some agent is pursuing a current line of reasoning which may be based on assumptions which are no longer valid. This gap between the model of the world assumed by an agent reasoning, and its current axiomatic description obviously weakens the meaning of a logic inference.

Since non-monotonic reasoning [14] intentionally falls out of the scope of this paper, this problem has been faced by adopting a simple classification scheme for the different sorts of knowledge involved. Take for instance the grid doors information represented by `door/3` facts in CU `grid` in the Mouse Agent Example. There, a logic inference would have sense at any time (also during speedy mouse exploration) since such information is stable, even though partial.[1] On the other hand, if more than one speedy mouse is activated, they would compete for cell occupation while exploring the grid. Competition is managed through a simple semaphoric protocol based on facts of the form `freeCell(X,Y)` to be withdrawn from `grid` before to move to cell `(X,Y)`, and then re-inserted when leaving it. Any (sequential) logic inference based on such kind of knowledge should then be avoided.

As a result, the $\mathcal{ACLT}$ model distinguishes between two sorts of knowledge which may happen to be involved in a communication via a CU:

---

[1] Here, we obviously lay on the implicit assumption that a faulty sensor does not produce any information at all. Conversely, we should take into account the case of false believes about the world, which would call for more complex approaches to non-monotonic reasoning.

– partial, incomplete knowledge
– transient knowledge

The former category refers to those parts of the application domain which are unknown yet, but that can be considered (practically) stable (such as the structure of the grid in the example). The latter refers to that part of the world which evolves during agent life (such as each cell being at a given moment free or occupied), so that they cannot be taken safely as a basis for logic operations.

A possible approach might exploit the multiplicity of "knowledge containers" (CUs) in order to keep transient knowledge separate from incomplete knowledge. In other words, each CU might be defined as containing either one category or the other one, thus forbidding *in* or *demo* class primitives, accordingly.

On the other hand, first-order logic-based languages provide another way to partition knowledge into chunks. In particular, predicate symbols can be thought as a source of a primary form of modularity: given predicate `p`, the collection of the facts of the form `p(t̃)` constitutes a unique knowledge chunk. Thus, one could define each predicate symbol as representing either a transient or a partial form of knowledge.

Logic theories (CUs, too) represent objects of the application domain, denoted by elements of the Herbrand Universe (ground terms of a logic language), according to a given interpretation. On the other hand, predicate symbols (which the Herbrand Base of a logic program is built from) define relations over computational objects. Since it seems more reasonable to associate information categories to relations rather than to objects, $\mathcal{ACLT}$ allows each predicate symbol occurring in a CU to be defined (either explicitly, through suitable static declarations, or implicitly, by the primitives dynamically used to access the corresponding axioms) as being either *transient* or *extendible*.

Thus, `door/3` predicate in the Mouse Agents Example is implicitly defined as *extendible* with respect to CU `grid`: any (exploring) agent can add further information about grid doors, and perform logic inference based on this knowledge: however, this knowledge cannot be removed by no one. This corresponds to an idea of the grid structure as being (relatively) non-mutable with respect to the mouse agents computation. Otherwise, it may happen the case of a lazy mouse reasoning on its path toward cheese, trusting some door knowledge, while this knowledge has meanwhile being removed by some other agent, making the assumption inconsistent.

At the same time, a competition between many speedy mouse agents, fighting for cell access, is based on a transient `freeCell/2` predicate. Grid structure is fixed, however one given cell being free or occupied by a mouse is an information changing as the multiagent system evolves. As a result, the interaction based on `freeCell` results in a sequence of *in* and *out* operations performed by different agents, while no agent activates an inferential activity based on sequential exploration and backtracking on `freeCell` axioms, which is quite reasonable. Thus, no `demo(freeCell(X,Y))@grid` is conceptually acceptable in the $\mathcal{ACLT}$ categorization.

4 Sketch of the $\mathcal{ACLT}$ implementation

$\mathcal{ACLT}$ has been built starting from a SICStus Prolog [6] system. Since it has been implemented as a library which can be loaded by any SICStus logic process, any SICStus program can easily become an $\mathcal{ACLT}$ agent.

Agents can be spread over a network of machines, each one running an $\mathcal{ACLT}$ daemon. Daemons are in charge of managing communication between $\mathcal{ACLT}$ processes running on the same machine (through UDP connections), as well as on different machines (through TCP connections), ensuring transparency of the communications: no information about physical allocations of agents is needed in order to access to any CU.

The whole $\mathcal{ACLT}$ system (both daemons and agents) is built by exploiting the full power of the CSM programming environment [7]. CSM actually implements upon SICStus Prolog (again, as a SICStus library) a model for declarative object-oriented languages based on first-order logic [8]. Thus, the $\mathcal{ACLT}$ kernel is defined as an open software layer which can be easily specialized by exploiting dynamic inheritance mechanisms. More refined communication protocols can be implemented on top of the $\mathcal{ACLT}$ basic system, by simply adding new layers. In addition, each $\mathcal{ACLT}$ agent can be developed according to an object-oriented design, by exploiting at the same time the full power of a declarative language.

Even though the SICStus programming environment (as well as CSM) is available on most hardware platform (UNIX, PC, Macintosh), current implementation works only on a network of Sun and HP workstations, and on PCs running Linux. Further work will be devoted to extend the number of the different architectures supporting the $\mathcal{ACLT}$ system.

5 Conclusions

Many different models for process communication/synchronization based on logic languages have been proposed in the literature (such as [5,9]). However, the original contribution of $\mathcal{ACLT}$ lays in the twofold interpretation of the communication units as both repositories for message exchange, and logic theories representing evolving models for the objects of the application domain. In fact, $\mathcal{ACLT}$ make such two readings coexist in the same conceptual framework, by clarifying the relationship between time-related tasks and logic inference. By exploiting a simple knowledge classification scheme which distinguishes between partial and transient knowledge, $\mathcal{ACLT}$ provides a set of disciplined primitives for logic inference, allowing multiagent systems to be designed where both single agents, and the system organization as a whole, are able to perform reasoning activities interleaved with reactive behaviours.

On the other hand, despite of some analogies, the $\mathcal{ACLT}$ model cannot be properly classified as a knowedge assimilation framework [12, 13], since the flow of input sentences (knowledge coming from agents) into a theory (CU) is subsantially independent of the logical relation between the theory and the knowledge to be assimilated. Moreover, the $\mathcal{ACLT}$ scheme does not provide direct support for non-monotonic reasoning [14]. Even though it may be argued that $\mathcal{ACLT}$ may

somehow be exploited in order to deal with uncertain knowledge, this issue falls far outside the scope of this work.

The starting point for the development of $\mathcal{ACLT}$, rooted in the CPU model [3], has to be found in the experimental work made with CARA [11] (where the Mouse Agents Example comes from), where the need for a unique model integrating reasoning and reaction clearly emerged. Multiagent systems built with CARA were meant to exploit the power of logic programming in advanced application domains such as robotics, where high-level symbolic activities have to coexist with low-level reactive behaviours. Further work will be then devoted to test the effectiveness of both $\mathcal{ACLT}$ model and implementation in real cases, such as the robotics case studies used for CARA testing.

Bibliography

1 R. Englemore, T. Morgan (eds.). *Blackboard Systems*. Addison-Wesley, Reading, Mass., 1988.
2 D. Gelernter. *Generative communication in Linda*. ACM Transactions on Programming Languages and Systems, 7(1), January 1985.
3 P. Mello, A. Natali. *Extending Prolog with Modularity, Concurrency and Metarules*. New Generation Computing, 10(4), August 1992.
4 A. Brogi, P. Ciancarini. *The Concurrent Language, Shared Prolog*. ACM Transactions on Programming Languages and Systems, 13(1), January 1991.
5 D. Gelernter. *Multiple Tuple Spaces in Linda.* Proceedings of PARLE, 1989, LNCS 365.
6 Swedish Institute of Computer Science. *SICStus Prolog User's Manual*. Kista, Sweden, 1994.
7 E. Denti, A. Natali, A. Omicini. *Moving Prolog Toward Objects.* In E. Tick, G. Succi (eds.), Implementations of Logic Programming Systems, Kluwer, Dordrecht (NL) 1994. pp. 89-102.
8 A. Omicini, A. Natali. *Object-Oriented Computations in Logic Programming*. In M. Tokoro, R. Pareschi (eds.), Object-Oriented Programming. LNCS 821. New York, Springer-Verlag 1994, pp. 194-212.
9 P. Ciancarini. *Distributed Programming with Logic Tuple Spaces.* New Generation Computing, 12, 1994.
10 F. Zanichelli, S. Caselli, A. Natali, A. Omicini. *A Multi-Agent Framework and Programming Environment fot Autonomous Robotics*. Proceedings of the International Conference on Robotics and Automation, ICRA '94, S. Diego, May 1994.
11 E. Denti, A. Natali, A. Omicini, F. Zanichelli. *Robot Control Systems as Contextual Logic Programs*. In C. Beierle, L. Plümer (eds.), Logic Programming: Formal Methods and Practical Applications. Elsevier, 1994.
12 R.A. Kowalski. *Logic without Model Theory*. 1993. Found at the following WWW location: http://src.doc.ic.ac.uk/ic.doc.lp/Kowalski/models.ps.gz.
13 R.A. Kowalski. *Logic for Problem Solving.* North Holland Elsevier, 1979.
14 R. Reiter. *Non-monotonic Reasoning*. In Ann. Rev. Computer Science, 1987, 2, pp. 147-186.