AN INTRODUCTION TO COORDINATION

Giuseppe Vizzari

CREDITS

- Most the material is adapted from Paolo Ciancarini's course on "Coordination Languages and Models", held at the EASSS 2001 PhD summer school, properly updated
- Original material can be found at http://www.cs.unibo.it/~cianca/wwwpages/seminari/lista.html

OVERVIEW

Background and motivation

What is a coordination language?

Coordination mechanisms in Linda and derived languages

COORDINATION

- Coordination is a key concept for studying the activities of complex dynamic systems

  Coordination is managing dependencies between activities

- Such a definition implies that all instances of coordination include agents performing activities that are interdependent [Malone and Crowston 94]

- Due to its fundamentality, this notion covers a lot of facets, for instance in distributed artificial intelligence, robotics, biology, and organisational sciences

- Here we see coordination from the viewpoint of modeling for supporting programming languages and software engineering

  Coordination is the process of building programs by gluing together active pieces [Carriero and Gelernter 92]

- Active pieces here can mean processes, objects with threads, agents, or whole applications

- programming = coordination + computation

COORDINATION PROGRAMMING

- Coordination programming [CarGel90] is "more natural" than sequential programming for applications requiring explicit parallelism
- To write a coordinated program:

1. choose the *conceptual class* that is most natural for the problem
2. write a program using the *software architecture* that is most natural for that conceptual class
3. if the resulting program is not acceptably efficient, *transform* it in a more efficient version by switching from a natural architecture to a more efficient one

Conceptual classes [Carriero Gelernter 1990]:

- coordination by result
- coordination by specialisation
- coordination by agenda

These classes differ in the starting approach to design a program to solve the problem:

- we can start from the intended result,
- or from the organisation of computing agents,
- or from the list of subtasks to be performed

EXAMPLE

Example: planning the building of a house

- we can decompose the intended final layout, separately building the components and then putting them together
- we can assign a special task to each available agent, aiming at exploiting each (specialist) agent in parallel given a list of building phases, or
- we can try to parallelize the building process

Interactive example: preparing the student's guide for a Master Degree course

COORDINATION BY RESULT

- The intended result of a program can usually be decomposed in several subresults; all the components of the result can then be processed separately and simultaneously
- We can design a parallel application around the data structure yielded as the ultimate result, and we get parallelism by computing simultaneously all the elements of the result
- *Result coordination* focuses on the shape of the finished product: usually it has to be a complex structure whose elements can be computed in parallel
- Typical examples:
  - When the program has to produce structured data and if we can specify precisely how each element of the resulting structure depends on the rest and on the input, then it is a good idea to attempt result parallelism

EXAMPLES OF COORDINATION BY RESULTS

Given 2 n-element arrays A and B, compute their sum S

Given 2 matrices M1 and M2, compute their product P

Sort a list using parallel merge sort

Additional examples?

COORDINATION BY SPECIALIZATION

Each available worker is assigned to perform one specified kind of work, and they all work in parallel (e.g. in pipeline) up to the natural restrictions imposed by the problem

We can plan an application around an ensemble of specialist programs connected into a logical network of some kind; parallelism results form all the nodes of the logical network being active simultaneously

Specialist coordination focuses on the makeup of the work crew (i.e. the "*software architecture*")

EXAMPLES OF COORDINATION BY SPECIALIZATION

A number of servers in an operating system;

A number of monitor/control processes in a realtime system

A parallel compiler built as a pipeline of fine-grain tools (eg. scanner, parser, code generator, optimizer)

Additional examples?

COORDINATION BY AGENDA (A)

Each worker is assigned to help out with the current item on the agenda, and they all work in parallel up to the natural restrictions imposed by the problem

We can plan an application around a particular agenda of activities and then assign several workers to each step

Agenda coordination focuses on the list of tasks to be performed; in this case, workers are not specialist: their structure is uniform (they input a task, solve it, and finally output the solution)

Two special cases of agenda coordination:

Data parallelism (synchronous)	Speculative parallelism (or-parallelism)

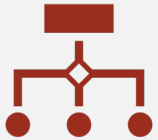
COORDINATION BY AGENDA (B)

Coordination by agenda involves a series of transformations to be applied to all elements of some set in parallel: typically we have a *master-worker* structure

A master process initialises the computation and creates a collection of identical worker processes; each worker is capable of performing any step in the computation; the master waits for solutions computed by workers

Workers repeatedly seek in the agenda a task to perform, get a task, perform the selected task, output the solution, and repeat; when no task remains, the program terminates

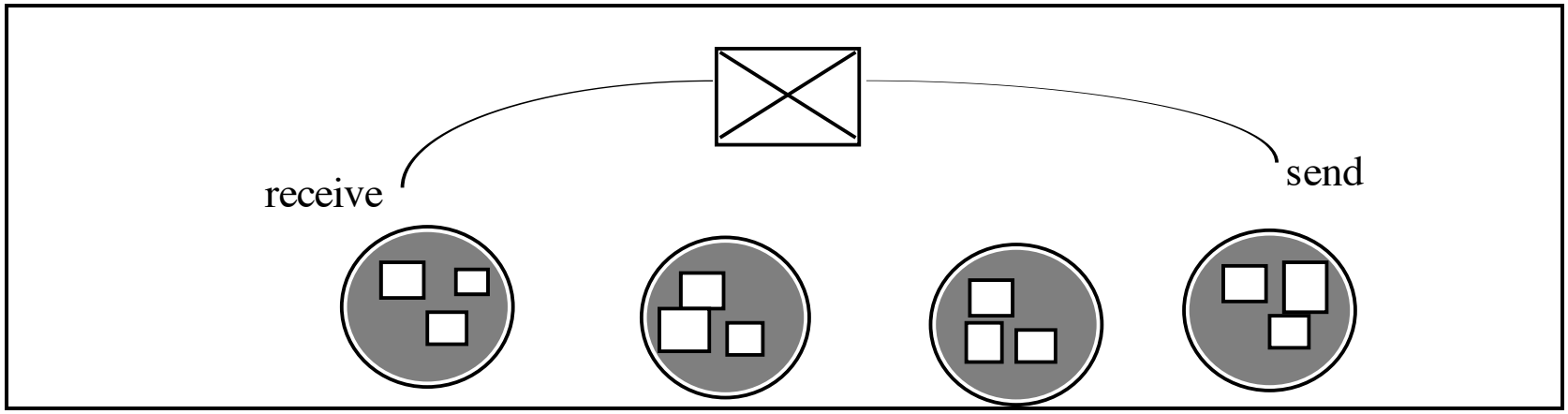
EXAMPLES OF COORDINATION BY AGENDA



A make utility which distributes sources to allow for parallel compilation

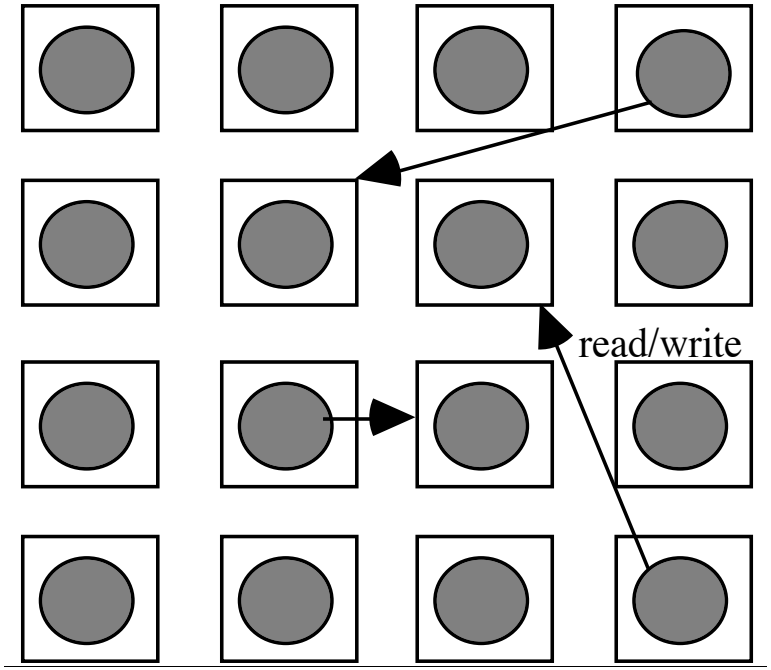
A chess program which searches in parallel the game tree

Additional examples?



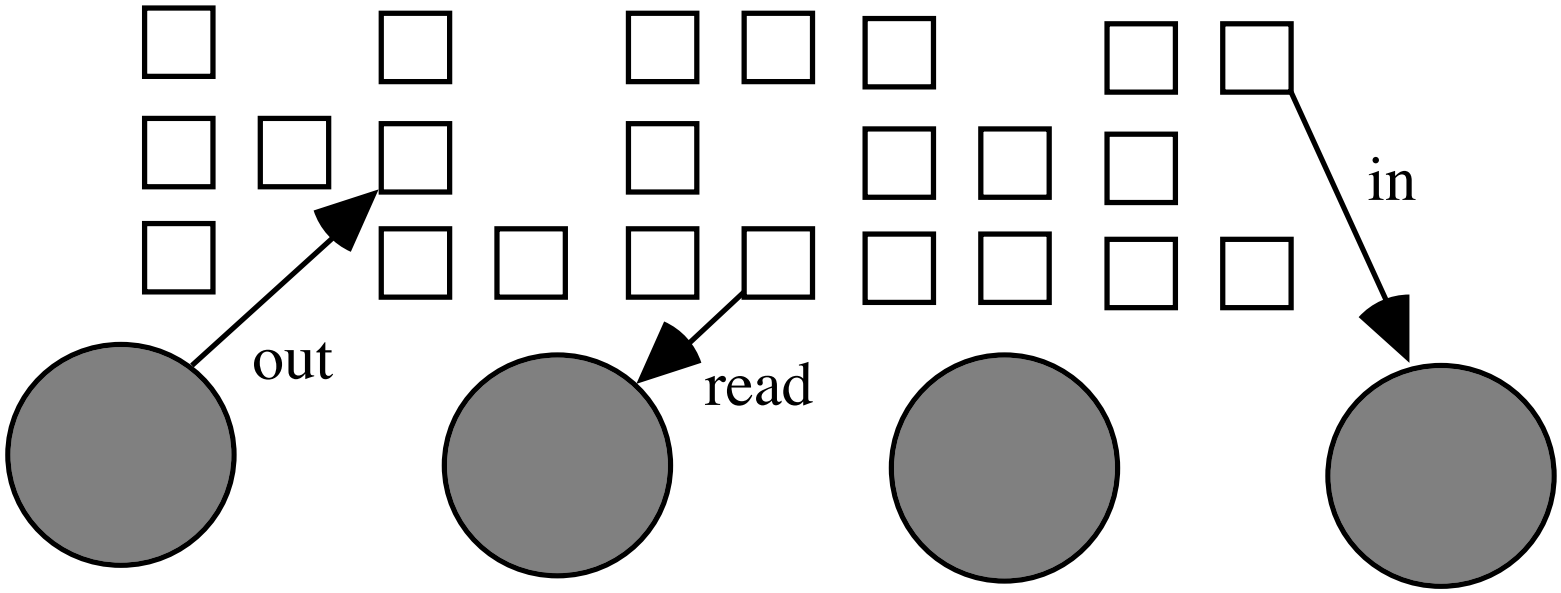
MESSAGE PASSING TECHNIQUES

- *Message passing* models allow to coordinate processes that can communicate with other processes through channels or ports on which messages are sent and received
- Typical languages of this class are Ada, CSP, Occam, POOL, concurrent logic languages, data flow languages
- This programming model is the basis of most operating systems architectures that use the client-server model
- It is also the basis of the actor (OO) model of computation



LIVE DATA STRUCTURES

- *Live data structures* coordination models allow to define data structures that contain active threads of computation; the threads can read/write other data structures under the control of other threads using synchronising primitives (e.g. semaphores, monitors, critical conditional regions, path expressions)
- Typical languages of this class are Concurrent Pascal, DP, Edison, Argus, Modula, Mesa, Concurrent Euclid, Unity (not *that* Unity!)
- Most ancient operating systems were designed using this kind of concurrency (e.g. Unix)



DISTRIBUTED DATA STRUCTURES (TUPLE SPACE)

- A *distributed data structure* is logically separated from processes that can manipulate it; in Linda, for instance, the distributed data structure is contained in a Tuple Space, that is a multiset of tuples; processes produce/consume tuples and create other processes
- Linda is the most known language that provides a distributed data structure; other languages that offer distributed data structures are Orca, Shared Prolog, Gamma

LINDA

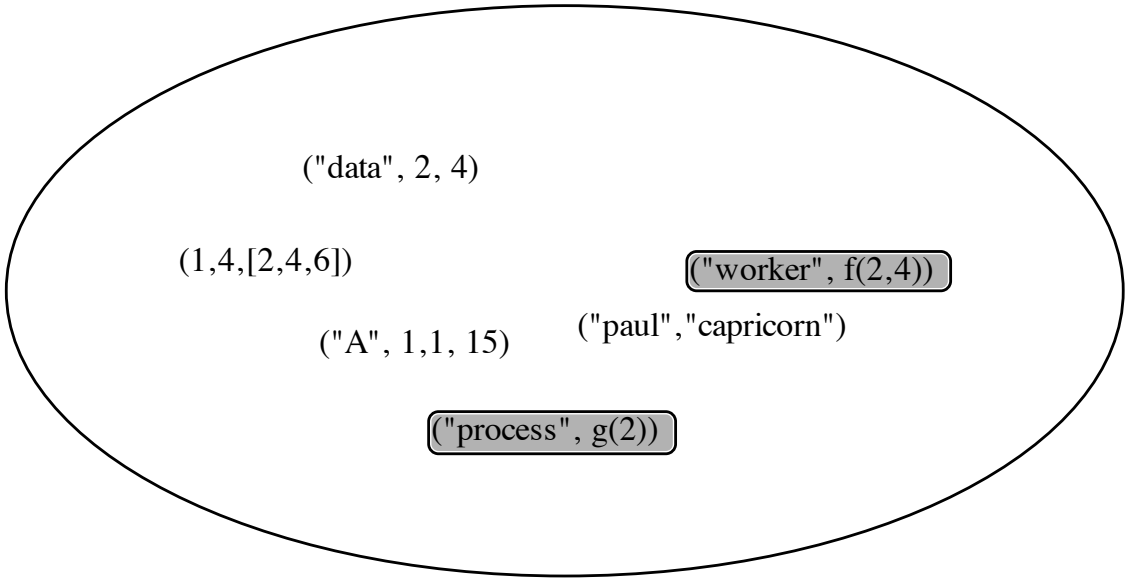
Linda consists of a few simple operations that have to be embedded in a host sequential language to obtain a parallel programming language

programming = coordination + computation
Linda introduced a new paradigm: *generative coordination*

A Linda program refers to a (physically distributed) data structure called *Tuple Space*, that is a multiset of tuples; there are two kinds of tuples:

passive tuples containing data	active tuples containing processes

A *tuple* is a sequence of typed *fields*



LINDA AS A COORDINATION MODEL

- The Tuple Space is a global computing environment conceptually including both data, in form of *passive* tuples, and agents, in form of *active* tuples
- All tuples are created by agents; they are atomic data (they can only be created, read or deleted)
- Agents cannot communicate directly:
- an agent can only read (`rd`) or consume (`in`) a tuple, write (`out`) a new tuple or create (`eval`) a new agent that when terminates becomes a data tuple

LINDA OPERATIONS

Tuples are created and manipulated by agents using the following operations:

- `out(t)` puts a new passive tuple in the Tuple Space, after evaluating all fields; the caller agent continues immediately
- `eval(t)` puts a new agent in the Tuple Space (each field containing a function to be computed starts a process); the caller agent continues immediately; when all active fields terminate the tuple becomes passive
- `in(t)` looks for a passive tuple in the Tuple Space; if not found the agent suspends; when found, reads and deletes it
- `rd(t)` looks for a passive tuple in the Tuple Space; if not found the agent suspends; when found, reads it
- `inp(t)` looks for a passive tuple in the Tuple Space; if found, deletes it and returns TRUE; if not found, returns FALSE
- `rdp(t)` looks for a passive tuple in the Tuple Space; if found, copies it and returns TRUE; if not found, returns FALSE

LINDA - MATCHING RULES

Operations in, read, inp, readp access tuples in the Tuple Space *associatively* (by pattern matching)

Their argument is *a tuple schemata*, namely a tuple containing formal fields used to search a tuple by pattern matching in the Tuple Space;

if a matching tuple is found, the operation is successful

LINDA EXAMPLES

Example:

```
out("string", 10.1, 24, "another
        string")

real f; int i;
rd("string", ?f, ?i, "another
        string")
```

succeeds

```
in("string", ?f, ?i, "another
        string")
```

succeeds

```
rd("string", ?f, ?i, "another
        string")
```

does NOT succeed

Example:

```
out(1,2)
rd(?i,?i)
```

does not succeed

Example:

```
eval("worker",7,exp(7))
```

creates an active tuple

```
in("worker",?i,?f)
```

succeeds when eval terminates

Example:

```
eval("double work", f(x), g(y))
in("double work", ?h, ?k)
```

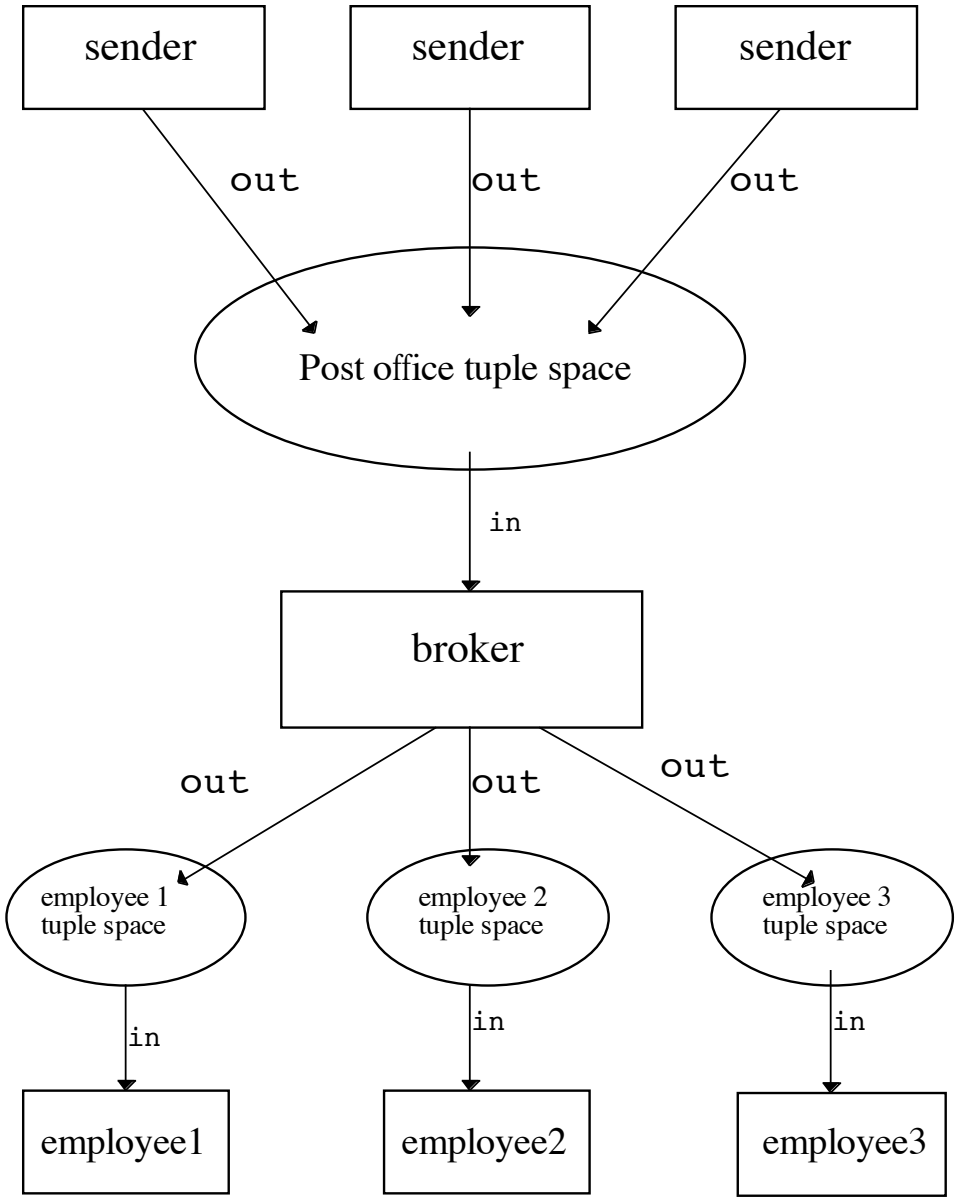
succeeds when both active fields terminate

EXAMPLE OF COORDINATION PATTERN: MASTER WORKER

```
master(){

for all tasks {
   /* build task structure for this
   iteration */

   ...
   out("task", task_structure);
   }

for all tasks {
   in("result",?&task_id,?&result_structure)
   ;
   /* update total result using this result
   */
   ...
   }

}

worker(){
   while(inp("task",?&task_structure){
   /*exec task*/
   ...
   out("result,task_id,result_Structure);

}
```

DINING PHILOSOPHERS

```
#define NUM 5
phil(int i){
            while(1) {
                        think();
                        in("room ticket");
                        in("fork", i);
                        in("fork", (i+1)%NUM);
                        eat();
                        out("fork", i);
                        out("fork", (i+1)%NUM);
                        out("room ticket");
            }
}


real_main() {
            int i;
            for (i=0, i<NUM, i++){
                        out("fork", i);
                        eval(phil(i));
                        if (i<(NUM-1)) out("room
ticket");
            }
}
```



ANOTHER EXAMPLE...

- What type of conceptual class of coordination is this?
- Would it be the same if the post office could manage different, heterogeneous requests?

WORK ON EXAMPLES

- Let's do some work together…
- Define coordination mechanism for exam grade registration…

LET US CONSOLIDATE A LITTLE BIT

WHAT IS A COORDINATION MODEL, REALLY?

- Historically, Linda was introduced as a new model for parallel programming, more flexible and high level w.r.t. its competitors (“Linda Is Not aDA”)
- It proved that it is possible to “think in a coordinated way” abstracting from low level mechanisms for concurrent, parallel, or distributed programming
- It showed that a careful design can avoid to pay a performance price to use a high-level coordination model
- Remember, “A coordination model is the glue that binds separate activities into an ensemble” [CarGel92]
- In other words, a coordination model provides a framework in which the interaction of individual agents can be expressed.
- This covers the issues of dynamic creation and destruction of agents, control of communication flows among agents, control of spatial distribution and mobility of agents, as well as control of synchronization channels and distribution of actions over time

LINDA AND OPEN SYSTEMS

- Whereas Linda was born for batch (single user) parallel programming, its underlying coordination model, namely the tuple space, has been investigated as a model for open systems design, because of the following features:
  - Uncoupling of agents
    Senders and receivers using the tuple space as channel/repository do not know about each other
  - Associative addressing
    Linda agents use patterns to access data, namely they say *what* data they need rather than *how* it should be found.
  - Non determinism
    Associative addressing is intrinsically non-deterministic, that is appropriate to manage information dynamically changing
  - Separation of concerns
    Linda was the first language to focus solely on coordination, proving that coordination issues are orthogonal to computation issues, that is useful to deal with legacy systems (seen as formed by components reusable and pluggable in novel software architectures)
- Whereas computation deals with *algorithms*, coordination deals with *architectures* (configurations of agents)

COORDINATION MODEL AND LANGUAGES (1/2)

- A *coordination model* offers mechanisms at least to control component generation/termination, (asynchronous) communication, multiple communications flows, multiple activity spaces; usually a coordination model consists of some coordination mechanisms that are added to a host language
- **Definition**:
  A *coordination model* is a triple (E,M,L), where:
  - E are the *coordinable entities* : these are the active agents which are coordinated. Ideally, these are the building blocks of a coordination architecture. (eg. agents, processes, tuples, atoms, etc.)
  - M are the *coordinating media*: these are the media enabling the coordination of interagent entities. They also serve to aggregate a set of agents to form a *configuration*. (eg. channels, shared variables, tuple spaces, bags)
  - L are the *coordination laws* ruling actions by coordinable entities (eg. associative access, guards, synchr. constraints)

COORDINATION MODEL AND LANGUAGES (2/2)

- In practice, a coordination language includes clearly defined mechanisms for communication, synchronization, distribution, and concurrency control
- It is based on a formally defined coordination model and it is "independent" from any sequential language; however, it should be easily embedded in any programming language
- Example
  - Linda ("shared tuple space") coordinable entities: active tuples (agents)
  - coordination media: tuple space
  - coordination laws:
    - non blocking out of passive tuples;
    - blocking read/in by pattern matching;
    - complex eval semantics

LINDA FORMAL SEMANTICS

- Linda was defined at Yale and implemented over several hw architectures without any formal semantics.
- This caused some problems in porting Linda programs from an implementation to another
- A formal semantics for Linda was firstly defined in Z [Butcher 91]. A formal semantics by Yale people was given in [Gelernter & Zuck 1997]
- [Ciancarini et al 1994] studied a number of formal semantics, aiming at comparing the expressivity of some well known concurrent semantic models as a tool for the design of alternative language implementations
- After having abstractly specified the Linda coordination model, we introduced, studied and compared the SOS, CCS, PetriNet, and CHAM semantics for Linda
- Some researchers in York used the CHAM semantics to approach and model multiple tuple space extensions
- [Busi et al 97] studied the Turing-equivalence of Linda

EXTENSIONS ON LINDA APPROACHES...

- The expressiveness of coordination media often needs to be tailored to the complexity and peculiarity of the specific coordinated system
- So, a number of Linda derivatives, e.g.
  - Law-Governed Interaction [Minsky and Ungureanu, 2000]
  - MARS [Cabri et al., 2000]
  - **ReSpecT** [Omicini and Denti, 2001]
- focus on the programmability of the tuple space, so as to
  - make it possible to explicitly express the rules of coordination
  - embed them within the coordination abstraction
- There, arbitrarily-complex coordination policies can be in principle associated to each and every coordination medium, which could be individually programmed so as to embed either global or local coordination policies, as required by the specific coordinated systems

EXAMPLE OF RESPECT

```
% a new order is placed
reaction(out(new order(Customer, [(Book,Seller)], Carrier)), (
    in_r(new_order(Customer, [(Book,Seller)], Carrier)),
    % generate a new transaction ID
    in_r(trans_id_counter(ID)), NextID is ID + 1, out_r(trans_id_counter(NextID)),
    out_r(order_info(ID, Customer, [(Book,Seller)], Carrier)),
    out_r(order_state(ID, ordering)),
    % update buyer activity working list
    out_r(working_list(buyer, ID, Book, Seller)) )).

% a book is ready at the seller's: execute dispatching
reaction(out(book_ready(ID)),(
    in_r(book_ready(ID)),
    in_r(order_state(ID, ordering)),
    out_r(order_state(ID, ready)),
    % update carrier activity working list
    rd(order_info(ID, Customer, [(Book,Seller)], Carrier)),
    out_r(working_list(carrier, ID, Carrier, [(Book, Seller, Customer)] )) )).

% the book has been collected from sender (seller)
reaction(out(book_dispatched(ID)),(
    in_r(book_dispatched(ID)),
    in_r(order_state(ID, ready)),
    out_r(order_state(ID, dispatching)) )).

% book delivered to customer: execute payment
reaction(out(book_delivered(ID)),(
    in_r(book_delivered(ID)),
    in_r(order_state(ID, dispatching)),
    out_r(order_state(ID, delivered)),
    % update payment activity working list
    out_r(working_list(payment, ID)) )).
```

(from [Ricci et al., 2001])

ADDITIONAL REFERENCES (IN ADDITION TO CIANCARINI'S FULL COURSE)

- Butcher, Paul. "A behavioural semantics for Linda-2." *Software Engineering Journal* 6.4 (1991): 196-204.
- Busi, N., Gorrieri, R., Zavattaro, G., & Zamboni, M. A. (1997). On the Turing equivalence of Linda coordination primitives. *Electr. Notes Theor. Comput. Sci.*, *7*, 75.
- Cabri, G., Leonardi, L., and Zambonelli, F. (2000). MARS: A programmable coordination architecture for mobile agents. IEEE Internet Computing, 4(4):26–35.
- Ciancarini, Paolo. "Distributed programming with logic tuple spaces." *New Generation Computing* 12.3 (1994): 251-283.
- Carriero, Nicholas, and David Gelernter. *How to write parallel programs: a first course*. MIT press, 1990.
- Gelernter, David, and Nicholas Carriero. "Coordination languages and their significance." *Communications of the ACM* 35.2 (1992): 96.
- Gelernter, David, and Lenore Zuck. "On what linda is: Formal description of linda as a reactive system." *Coordination Languages and Models*. Springer Berlin Heidelberg, 1997. 187-204.
- Malone, Thomas W., and Kevin Crowston. "The interdisciplinary study of coordination." *ACM Computing Surveys (CSUR)* 26.1 (1994): 87-119.
- Minsky, N. H. and Ungureanu, V. (2000). Law-Governed interaction: A coordination and control mechanism for heterogeneous distributed systems. ACM Transactions on Software Engineering and Methodology (TOSEM), 9(3):273–305.
- Omicini, A. and Denti, E. (2001). From tuple spaces to tuple centres. Science of Computer Programming, 41(3):277–294.
- Alessandro Ricci, Enrico Denti, Andrea Omicini:Agent Coordination Infrastructures for Virtual Enterprises and Workflow Management. CIA 2001: 235-246