Lecture 3: Coordination languages

Contents

• Beyond Linda

• Coordination and declarative programming

• Coordination and OO programming

• Coordination languages for mobile agents
Linda and friends

What Linda taught:

programming =
(distributed) coordination + (sequential) computation

The basic idea is to give a small set of coordination primitives useful to design a distributed system

The main advantage consists of automatic load balancing implicit in agenda parallelism (associative matching) and simplicity of master/worker software architectures exploited for parallel programming

Moreover, Linda is intrinsically multiparadigm and can support interoperability across languages

However:

• no formal semantics
• no logic to reason on coordination
• no general hints on how to embed a coordination model in a host language
• few hints on how to optimize coordination-level language mechanisms independently from the underlying hardware
Marrying Linda with a host language

Linda is a coordination language, that is it needs to be embedded in a host sequential language to offer a full (parallel) programming language.

Linda has been coupled to several conventional languages, like C and FORTRAN (original SCA™ Linda), Pascal, Scheme, Ada, Eiffel, Prolog, etc., and implemented on several hw architectures.

Remark: For the Linda family of languages a semantics gap existed: they were designed and implemented for several distributed architectures, but not formally defined; the Linda concept itself has been instantiated in different forms by different language designers.
Language design issues

The main problems to be solved by a language designer are:

⇒ Data structures and types
   Which data structures are allowed in the tuples? (for instance, in C-Linda pointers are not allowed)

⇒ Matching rules for operations in the Tuple Space
   How is defined the matching operation? (for instance, in Prolog-Linda it has to be based on unification)

⇒ Control of tuple operations
   Which kind of control structures can be used with Linda operations? (for instance, backtracking is difficult to implement in a distributed setting)

⇒ Definition of eval
   Which is the semantics of new processes/agents? Which code and data of the parent process they can refer to? How many active threads can be included in a tuple? How do they synchronize?
A universal Linda interpreter

In Yale, each Linda implementation has been studied and optimized for a specific language/hw architecture; however, a more universal although naive approach can be used.

In fact, a simple Linda implementation is easily obtained using a client-server sw architecture, which also allows multi-language interoperability of heterogeneous hosts.

Programs which wish to use the tuple space are clients of servers which implement the Tuple Space; in [Pinakis 93] there are two types of servers (a pair of such servers has to be present each node):

• *tuple servers*, which manage a *partition* of currently active tuple types
• *type servers*, which are used to locate tuples of a particular tuple type

Each tuple type is managed by the tuple server on exactly one host (owner-id) and has a unique tuple-id on that host.

Performing a Linda operation therefore involves two steps:
1. determine (owner-id,tuple-id)
2. send request to owner host and possibly wait for reply
Pinakis’ solution

The following diagram explains a sw architecture defined by Pinakis

Note: since eval operation can easily simulated by in/out/rd and “demon” workers, it is not implemented
Linda and logic programming

There are several ways to define a Linda-based logic language

a) Prolog + non-backtrakable Linda primitives: simple implementation, inconsistent with the “pure logic programming paradigm” (but similar to assert/retract) [SICSTUS, Tarau’s BinProlog];

b) Prolog + backtrakable Linda primitives: complex implementation (distributed backtracking) similar to DeltaProlog [Monteiro-Porto]; intensional or extensional tuple space [Delta-D-Prolog: Sutcliff and Pinakis 91, Omicini 97]

c) A parallel logic language based on multiset rewriting, i.e. a new parallel logic language with “Linda flavor” [Shared Prolog; μLog; Gammalog]

d) Using non standard semantics, like linear logic [LO by Andreoli; Forum by Miller]
Sicstus™ Prolog-Linda

Sictus Prolog offers some simple Linda-like predicates

One Prolog process is running as a server and one or more processes are running as clients; all processes are communicating via sockets.

The server is in principle a blackboard on which the clients can read/write using the Linda operations:
- `out/1`
- `read/1`
- `in/1`

If the data is not present on the blackboard, the predicates suspend the process until they are available.

There are some more predicates: `in_noblock/1` and `rd_noblock/1` do not suspend if the data is not available—they fail instead.

A fetch of a conjunction of data is done with `in/2` or `rd/2`:

\[
\text{in}([e(1), e(2), \ldots, e(n)], E) \\
E \text{ instantiated to the first tuple to become available, among those listed}
\]

EXAMPLE: A simple producer-consumer

```
producer :- % client1
  produce(X), out(p(X)), producer.
produce(X) :- ..... 

consumer :- % client2
  in(p(A)), consume(A), consumer.
consume(A) :- ..... 
```
MultiProlog

The blackboard interpretation of logic programming is based on a Linda-like coordination model.

MultiProlog (DeBosschere & Tarau) includes blocking and non-blocking built-ins to manipulate distributed blackboards:

\[
\text{out}(X) :- \text{tell}t(\text{world}, X).
\%
\text{puts } X \text{ on the server}
\]

\[
\text{in}(X) :- \text{get}t(\text{world}, X).
\%
\text{takes a matching term } X \text{ from the server}
\]

\[
\text{all}(X, Xs) :- \text{tell}t(\text{world}, X).
\%
\text{puts } X \text{ on the server}
\]

\[
\text{out}(X) :- \text{tell}t(\text{world}, X).
\%
\text{puts } X \text{ on the server}
\]

MultiProlog has been used to build a prototype MUD (LogiMOO) and some prototype applications in the field of electronic commerce [Tarau and others, 1997]
The tuple space coordination model in Shared Prolog

Basic idea: a parallel logic language based on the Tuple Space concept, i.e. a logic language with “Linda flavor”

- *Prolog for sequential computation*
- *shared-dataspace of logic terms*
- *rewriting rules and unification*

A logic tuple space implements a blackboard-like data structure that can be shared by a number of logic processes (Prolog programs)

Shared Prolog (SP) evolved as “kernel language” of a family of coordination languages, all based on extensions of the basic tuple space model.

Extended Shared Prolog (ESP): extends SP with multiple tuple spaces, to design logically distributed systems

SP vs Linda:
- SP uses full logic unification, Linda uses (sequential) typed pattern matching
- In Linda `rd`, `in` and `out` are allowed everywhere in a program; SP is more structured and allows transactions on tuple space
Shared Prolog

A SP program is composed of a set of theories and a dataspase (or blackboard), i.e. a (possibly empty) multiset of Prolog atoms.

For instance, suppose we have to specify a reservation system including a database (stored in the blackboard), a set of theories corresponding to the agencies, and a theory corresponding to the airline company.

The system is activated by an initial goal:
\[
database \parallel \text{airline} \parallel \text{agency}_1 \parallel \ldots \parallel \text{agency}_n
\]

The symbol “||” stands for parallel execution, i.e. this goal is composed of n+2 AND-parallel processes.

One of the atoms in the initial goal must correspond to a blackboard: in this example it is the database.

Here the blackboard initially contains the number of free seats for each flight; its initial configuration is given by the following rule:

\[
database:- \{ \text{flight}(1,p_1), \ldots, \text{flight}(k,p_k) \}.
\]

where in \text{flight}(i,p_i) i is the flight number and \(p_i\) the number of free seats.
A theory

\texttt{agent\ agency_i:-}

\texttt{eval}

\texttt{not\ ready(agency_i)}

\texttt{| \hspace{1cm} initialization} \texttt{\{ready(agency_i)\}}

\texttt{\{ready(agency_i)\}}

\texttt{| \hspace{1cm} query\ to\ the\ system} \texttt{read(C),exec(C,Ci)} \texttt{\{query(Ci)\}}

\texttt{\{result(agency_i,R)\}}

\texttt{| \hspace{1cm} answer\ from\ the\ system} \texttt{write(R) \{ready(agency_i)\}}

\texttt{\textbf{with}exec(reserve(F,N),\textbf{exec}(F,N,agency_i))\textbf{.exec}(info(F),info(F,agency_i))}. 
Informal semantics

Each agent is able to perform some operations on tuples

• associative (blocking) test of a tuple contained in the dataspace; in SP the test operation has a broader semantics than read in Linda: a number of predefined tests on the dataspace are allowed, depending on the chosen type system for tuple arguments.

• associative (blocking) consumption of a tuple contained in the dataspace; this operation also has a broader semantics than in in Linda: several tuples can be collected and consumed by special operators that return multisets.

• asynchronous creation of a tuple inside the dataspace.

In SP there is no need of an operation like eval for creating active tuples because we adopt a "chemical" semantics, that is tuples that match coordination rules are implicitly active.
% program to find the best path in a DAG
% (it is acyclic!)

[in(s,0), in(a,1), in(b,1), in(c,1), in(d,2),
in(e,3), in(f,1), in(g,1), in(h,3), in(i,2),
in(j,1), in(k,2), in(l,2), in(m,1), in(t,4),
e(s,a,5), e(s,b,6), e(s,c,4), e(a,d,2), e(a,e,5),
e(b,d,1), e(b,e,3), e(c,e,4), e(c,f,1), e(d,g,7),
e(d,h,2), e(e,h,1), e(e,i,3), e(f,h,2), e(f,i,2),
e(g,j,4), e(g,k,2), e(h,k,3), e(h,l,2), e(i,l,4),
e(i,m,3), e(j,t,5), e(k,t,1), e(l,t,4), e(m,t,2),
b(s,s,0)] @ best.

agent best :-
eval
  b(X,A,C1), [e(A,B,C2)] % guard
  | % commit
  NewC is C1+C2, % Prolog goal
  [p(X,B,NewC)] % out
  #
  [p(A,B,C1), in(B,N)], p(A,B,C2) C1 >= C2
  |
  DecrN is N-1,
  [p(A,B,C2), in(B,DecrN)]
  #
  [p(A,B,C), in(B,1)]
  |
  [b(A,B,C)]
Implementing Shared Prolog

a) Distributed workers.

b) Distributed agents.

c) Distributed rules.
Implementing Shared Prolog

Theory 1 agent

... 

Theory k agent

Blackboard agent

Agent (extended Prolog process)

Interprocess communication

Blackboard manager

Theory 1

Theory n

C code for send & receive

C code for send & receive

C code for send & receive

Global communication server
Analyzing coordination in SP

Flow Graphs of SP Programs

Let $a$ be an atom, $P_1$ and $P_2$ be patterns

**OUTFLOW**
has an edge labeled with $a$ if
$a \in \text{Out}(P_1)$
AND
$(a \in \text{Read}(P_2) \text{ OR } a \in \text{In}(P_2))$

**INFLOW**
has an edge labeled with $a$ if
$a \in \text{In}(P_1)$
AND
$(a \in \text{Read}(P_2) \text{ OR } a \in \text{In}(P_2))$

**CONFLICTFLOW**
$P_1$ and $P_2$ are connected if
they belong to the same theory
OR
$\text{In}(P_1) \cap \text{In}(P_2) \neq \emptyset$
Extended Shared Prolog (ESP)

Extended Shared Prolog adds (nested) multiple ts to SP

A run-time for ESP

system

name_server

empty_mcb(mcb(3),[mcag(4),mcag(5)])
root_of_mcb([c,b,a],mcbb(0),[mcag(1),mcag(2)])
Linda and oo programming

In [MatsuokaKawai 88] we find the first proposal for an OO Linda, based on a reformulation of the TS coordination model for OO programming in SmallTalk.

- tuples and tuple spaces become objects; an object now can retain a msg/tuple even if does not have the ability to understand it, or can transparently delegate it.

- sender and receiver have to be symmetrical (in Linda they are not, because all the receivers must share the same info on shared tuples): so both out tuples in TS, and in this model we have both receiver and sender tuples.

- logical distribution of tuple spaces, to improve efficiency and security, and offer naming scope to tuples.

Class Tuple (Tuple)
a tuple is an arbitrary sequenced collection of element objects, including tuple spaces; matching in complicated by inheritance.

Class Formal (Formal arguments)
this is only used in communication matching.

Class TS(Tuple Space)
- put:<Tuple> like out in Linda
- set:<Tuple> synchronous out
- get:<Tuple> like in in Linda
- read:<Tuple> like read in Linda

The model also introduced operations to manipulate multiple tuples (bulk operations).

[Jellinghaus 90] introduced Eiffel Linda.
Objective Linda

Objective Linda [Kielman 97] introduces a concept of Object Interchange Language, a language independent notation to describe abstract data types.

An *OIL_object* defines the basic operations needed by all types: for instance, there is a match predicate which takes as parameter an object of the same type and decides if it matches some requirements.

Operations belonging to the coordination model are based on the types interfaces: a reader has to specify the type of object it wishes to get from the object space.

```lisp
class BARRIER_SYNC inherit OIL_OBJECT
redefine match end

creation create

feature barrier_id : STRING;
    create (id: STRING) is
        do barrier_id := id; end;
    match (candidate : like current): BOOLEAN is
        do Result :=
            candidate.barrier_id = barrier_id; end;
    end
```
Objective Linda

Since Objective-Linda is intended as a tool for designing open systems, all operations include a time-out; moreover, `out` and `eval` return a boolean value to signal successful completion.

```plaintext
out(m:MULTISET; timeout:REAL):BOOLEAN
eval(m:MULTISET; timeout:REAL):BOOLEAN
```

`in` and `rd` take as parameter the object (`OIL_OBJECT`) to search; the operation has to find at least `min` such objects and return `max` of them by the timeout.

```plaintext
in(o:OIL_OBJECT;min,max:INTEGER;timeout:REAL):MULTISET
rd(o:OIL_OBJECT;min,max:INTEGER;timeout:REAL):MULTISET
```

<table>
<thead>
<tr>
<th>min</th>
<th>max</th>
<th>timeout</th>
<th>behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>out</td>
<td>0</td>
<td>0</td>
<td>immediately fail on errors</td>
</tr>
<tr>
<td></td>
<td></td>
<td>t</td>
<td>wait t secs before failing on errors</td>
</tr>
<tr>
<td></td>
<td></td>
<td>infinite</td>
<td>Linda out</td>
</tr>
<tr>
<td>eval</td>
<td>0</td>
<td>0</td>
<td>immediately fail on errors</td>
</tr>
<tr>
<td></td>
<td></td>
<td>t</td>
<td>wait t secs before failing on errors</td>
</tr>
<tr>
<td></td>
<td></td>
<td>infinite</td>
<td>Linda eval</td>
</tr>
<tr>
<td>in</td>
<td>0</td>
<td>0</td>
<td>nay empty op</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>Linda inp</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>infinite</td>
<td>Linda in</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>n</td>
<td>any consume up to n objects</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>infinite</td>
<td>any consume all matching objects</td>
</tr>
<tr>
<td></td>
<td>infinite</td>
<td>any</td>
<td>t</td>
</tr>
<tr>
<td>rd</td>
<td>0</td>
<td>0</td>
<td>any empty op</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>Linda rdp</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>infinite</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>n</td>
<td>any consume up to n objects</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>infinite</td>
<td>any consume all matching objects</td>
</tr>
<tr>
<td></td>
<td>infinite</td>
<td>any</td>
<td>t</td>
</tr>
</tbody>
</table>
Multiple tuple spaces in Objective Linda

Configurations in Objective Linda include object spaces and OIL objects both passive and active (agents).

Agents by default can access two special object spaces:

- their context, which is the space inside which the eval op for such an object has been issued
- their self, which is a space directly associated to the object

Other object spaces are accessible through *logicals*, which are references which can be passed around by active objects.

When an object intends to let another object to access a space, it outs its logical to another space, from where it can be retrieved with a special `attach` operation:

```
attach(o:OS_Logical; timeout:Real): Object_Space
```

Formal semantics for Objective Linda has been studied in [HolvoetKielmann97] using a special Petri Net semantic model.
Programming with LO

A safe queen program

Coordination code

queens(N) <-
    board(N, 0, []).

board(N, N, B) @{remote_format('Safe board:~q~n', [B])} <-
    top.

board(N, I, B) @{check_new_boards(N, B, LB), I1 is I+1} <-
    make_boards(N, I1, LB).

make_boards(N, I, []) <-
    top.

make_boards(N, I, [H|T]) <-
    make_boards(N, I, T) &
    board(N, I, H).

Prolog computation code (fragment)

check_new_boards(N, B, LB) :-
    new_boards(N, B, LB1),
    safe_boards(LB1, LB).

ew_boards(0, _, []).

new_boards(X, B, [NB|LB]) :-
    X1 is X-1,
    append(X, B, NB),
    new_boards(X1, B, LB).

safe_boards([], []) :- !.

safe_boards([H|T], [H|T1]) :-
    safe(H),
    safe_boards(T, T1), !.

safe_boards([_H|T], T1) :-
    safe_boards(T, T1), !.
Programming with LO

TRAVEL AGENCIES

agency
client_req(spec,time,4)
...

agency
...

agency_process(spec)
criterion(time)
_nseat(4)

agency_collect(q230)
criterion(time)
_nseat(4)

agency_try(AA802)
nseat(4)
AA802:BA105:TWA732

agency_wait(q934,AA802)
nseat(4)
AA802:BA105:TWA732

agency_try(BA105)
nseat(4)
BA105:TWA732

agency_wait(q937,BA105)
nseat(4)
BA105:TWA732

AIRLINE COMPANIES

flight(TWA732,15)
flight(TWA1562,72)
flight(AA802,3)
flight(AA726,52)
flight(BA105,15)
flight(BA420,76)

info(q230,spec)

agency_try(BA105)
agency_wait(q937,BA105)
accepted(q937)
rejected(q934)

client_req(spec,time,4)
criterion(time)

agency_collect(q230)
offer(q230,TWA732)
offer(q230,AA802)

book(q934,AA802,4)

book(q937,BA105,4)

3 Coordination languages
Programming with LO

A customer has to buy and sell stocks. There are 3 stock markets: Tokyo, Paris, NewYork. To initialize them, we insert in each one some buy and sell offers. These offers are contained in the "stock_prices" list. There is a manager waiting for an order. When an order from a client arrives, the manager takes care of it and a new manager is created to receive other orders.

```
main @ client_list(Buy,Sell,I,B) <-
    stock_market(tokyo) @
    buy(sony,110,99,john,tokyo) @
    buy(honda,500,300,mary,tokyo) @
    sell(toshiba,100,50,mary,tokyo) @
    sell(sony,115,25,jim,tokyo) &

stock_market(paris) @
    buy(bull,90,876,mary,paris) @
    buy(lacoste,523,391,john,paris) @
    sell(renault,150,643,susan,paris) &

stock_market(newyork) @
    buy(sony,200,300,morris,newyork) @
    sell(renault,120,400,mary,newyork) &

manager @ client_list(Buy,Sell,I,B) &

counter.
```

Brokers need counter to avoid waiting indefinitely offers from stock markets. When the counter receives a "begin(I,N1)" message (I is the broker name, N1 is the name of the stock type) from a broker, it acts as a timer for him and a new counter is created to satisfy other requests from other brokers. When the fixed time is expired, counter sends a "end(I,N1)" message to the broker I and ends his work.

```
counter @ begin(I,N1) <-
    counter(I,N1) @ timer(20000) & counter.

counter(I,N1) @ timer(X) @ {X>0, Y is X-1} <-
    counter(I,N1) @ timer(Y).

counter(I,N1) @ timer(0) @ ^end(I,N1) <- #t.
```
LO has been implemented as a coordination language combined with host languages like Prolog and Modula-3

LO (like Linda) is not suitable for open systems design, because the program should be completely known before execution to be statically analyzed.

The main coordination medium in LO was the Forum, a sort of tuple space used for broadcast.

This coordination medium was abstracted from LO and used as basic coordination model of ForumTalk [Andreoli 96], a coordination platform to which LO programs could plug in at run time.

The architecture of ForumTalk consists of a coordination service implemented by several servers; the servers maintain a big LO proof-tree to which the servers can dynamically connect by the following ops:

- **forum-join**: register the server in the session; the tree is expanded at that node
- **agent-start**: starts an agent (a normal LO program) on the server where it is invoked
Coordination Language Facility (CLF)

The transactional nature of multiset rewriting as embedded in the LO model is at the basis of CLF [Andreoli et al. 1996], a software platform that merges OOP and transaction systems to coordinate access to distributed resources

• objects can dynamically offer new services
• multiparty negotiation can be specified and implemented

The sw architecture of a CLF application includes a number of application servers (participants) which are coordinated by one or more rule-based clients (coordinators) which execute LO programs
CLF basic coordination protocol

The CLF platform is implemented as an extension of client/server plus the following protocol:

**Inquiry**: (synchronous) a client inquires if a server holds or may produce a resource; the server (starts a thread and) answers with a stream of actions it may perform on demand to make such a resource available.

**Reservation**: (synchronous) the client requests the server to reserve one of the actions returned during the Inquiry phase. Accepting a reservation means that the server commits to perform the action on demand, with no possibility of failure.

**Confirmation/cancelation**: (asynchronous) The client may either confirm or cancel an action it has reserved. If confirmed the action must be executed leading to the deletion of the corresponding resource.

**Insert**: (asynchronous) the client requests the insertion into the server of a resource.
Structure of a CLF object

A CLF object can be implemented in any language, and can encapsulate any kind of resources

<table>
<thead>
<tr>
<th>Communication infrastructure (eg. CORBA, HTTP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>methods (standard oo protocol)</td>
</tr>
<tr>
<td>bank</td>
</tr>
</tbody>
</table>

resources

The interface of the object defines the properties (services) which are visible to the clients. Each service is implemented by a bank, which may encapsulate as wrapper a legacy application

The CLF library provides classes of objects in the target languages (currently C++ and Python)
Example

This example extends the flight reservation service already described for LO

\[
\text{journey_request}\text{(Name,From,To)} @ \\
\text{flight_from}\text{(From,TransferAt,FlightNum1)} @ \\
\text{flight_to}\text{(TransferAt,To,FlightNum2)} @ \\
\text{book_seat}\text{(Name,FlightNum1)} @ \text{book_seat}\text{(Name,FlightNum2)} <= \text{print_ticket}\text{(Name,FlightNum1,FlightNum2)}
\]

This coordination rule involves three participants:
- a user terminal which holds the tokens \text{journey_request} and \text{print_ticket}
- a flight database which holds the tokens \text{flight_from} and \text{flight_to}
- a travel agency which holds the tokens \text{book_seat}

signatures:
\[
\text{journey_request}\text{(Name,From,To)}:->\text{Name,From,To} \\
\text{flight_from}\text{(From,To,FlightID)}:\text{From}->\text{To,FlightID} \\
\text{flight_to}\text{(From,To,FlightID)}:\text{From,To}->\text{FlightID} \\
\text{book_seat}\text{(Client,FlightID)}:\text{Client,FlightID}->
\]

interfaces:
\[
\text{journey_request} = \text{user_terminal} \\
\text{flight_from} = \text{all_flight_info} \\
\text{flight_to} = \text{all_flight_info} \\
\text{book_seat} = \text{travel_agent}
\]
Actor Spaces

The ActorSpace model [Callsen Agha 94] is a model which extends actor-style point-to-point asynchronous communication with pattern-directed invocation and broadcasting.

The target systems to be designed are open, meaning that clients cannot be trusted, so security must be enforced.

**Client:** requests service from a server  
**Server:** provides a service to a set of clients  
**Manager:** surveys the system and adjusts it to suit needs as they arise.

ActorSpace (AS) adds new concepts to Actors:

- **attributes,** i.e. patterns that abstractly describe an external view of an actor; pattern matching can be used to pick actors whose attributes satisfy a given pattern. Thus in AS two forms of addressing coexist: an actor may be accessed by name or by pattern matching.
- **actorSpaces:** a passive container of actors, i.e. a scoping mechanism for pattern matching; actorSpaces can be nested. Actors and ActorSpaces may be made visible or invisible in an AS. A manager is associated to each AS, that validates capabilities and enforces visibility changes.
- **capabilities:** provide secure access control.

Communication operations:
- `send(pattern@actorSpace, message)`  
  (one to one)
- `broadcast(pattern@actorSpace, message)`  
  (one to many)
Coordination languages for the WWW

The WWW was born as a (huge) distributed hypertext document management system, but it rapidly become something more, and it is still evolving.

The availability of a Java VM on most Internet hosts offers a world-wide programmable infrastructure (a cyberspace?)

When a browser downloads and executes a Java applet this is mobility of code.

Some applications require complex coordination management:
- “intelligent” homes with networked appliances
- software distribution and maintenance
- infotainment (e.g. MUD, Internet playing clubs)
- electronic commerce
- workflow systems for active docs (eg based on XML/Java)
Jada [Ciancarini & Rossi 1996] is a combination of Java with Linda.

Jada provides:

• mobile object coordination: no syntax extension for Java, just a set of classes. Each data type used by Jada is a Java object.

• dynamic tuple creation: being an object, a tuple can be created with `new` and many different constructors are provided in order to build a tuple from a string, an array of objects or a set of arguments.

• multithreading support: in Jada different threads can access the same tuple space; blocking requests are managed at thread-level.

• open systems support: at any time threads or applications can perform an operation on a tuple space.

• associative access to collections of tuples: we provided special arguments for `in` and `read` requests (*any*) in order to allow the use of multiple matching tuples.

• no `eval`
Jada design choices

We designed Jada as a minimalist language, aiming to simplicity rather than performance

- The basic objects used in Jada are Tuple and TupleSpace
- To allow remote access to a tuple space the TupleServer and TupleClient classes are also provided.

TupleServer is a multithreaded class which translates requests received from instances of the TupleClient class in calls to the methods of the TupleSpace class

A TupleClient interfaces with a remote TupleServer object (which really holds the tuple space) and asks it to perform the Linda-like operations and to return the result

Both TupleServer and TupleClient are based on TupleSpace:

- TupleServer includes a tuple space and uses it to perform requested operations;
- TupleClient differs from TupleSpace: the access to the tuples is transformed in requests for the TupleServer

The Jada objects TupleServer and TupleClient communicate using sockets. TupleClient needs to know host and port of TupleServer, so a set of constructors is provided to specify TupleServer host and port
SHADE (Shared Dataspaces, DSI/Bologna) is an object-based coordination language extending Jada with transactions; a SHADE program consists of a set of classes; each class has the following form:

```
class C_name {method_1 # … # method_n}.
```

Method ::=  
- **read tuples**;  
- **in tuples**; % a multiset of tuples  
- **with** `{seq_program}`; % Java, C, Prolog  
- **send** tuples; % multicast  
- **create objects** % an object is a named multiset  
- **out** tuples. % terminate, or tuples to be written

When a program is run, one prototype for each class is created; objects (named multisets of tokens) are created by prototypes; a number of servers are able to get and manipulate objects-as-multisets.
**Mobility by migration**

Macondo [CiaRos99] is a Linda-based coordination environment where Java agents can migrate.

The coordination media consist of a set of distributed object spaces (similar to tuple spaces, but tuples are Java objects); agents can migrate from a space to another one.

An object space is used to coordinate mobile agents, namely to manage their relationship with other agents in the same place and with the underlying runtime system.

Agents at any time reference the local object space using the following operations:
- `in`, `read`, `out` an object in the local object space
- `register`, to request the runtime to issue a notification when some specific object appears in the object space
- `go`, to migrate to another place

Some manager agents can create and destroy places.

We are using Macondo for groupware for mobile users.
LIME

LIME (Linda in a Mobile Environment) [PMR99] adopts the view that physical and logical mobility coincide. A wireless network includes mobile hosts; each host can be the home of one or more mobile agents.

Each agent owns a local tuple space; when the host containing the agent joins a network, the local tuple space becomes public and it is “merged” with the global tuple space formed by spaces of agents already included in the network.

The global tuple space is thus “transient”: if an agent leaves the network, all “its” tuples not already consumed leave as well.
Conclusions

Linda’s tuple space is a simple embodiment of a concept of software bus, namely an interoperability platform (cf CORBA)

The tuple space can be implemented in several ways; an optimized implementation needs a static analysis phase

Marrying Linda with a conventional language needs a study of data structures used for coordination entities and media

The combination of Linda with logic languages has been studied especially from the point of view of parallel programming: large grain parallelism is especially suited to logic programs

The combination of a coordination model with object oriented languages is quite promising; still, a lot of research effort is necessary to understand the coordination requirements of distributed objects and define suitable coordination languages

The combination of coordination technologies with WWW is especially promising because they support in a very natural way concepts like open-endness and agent migration