Lecture 2: Coordination models

Contents

• Coordination models

• Coordination models based on multiset rewriting

• Coordination models based on multiple tuple spaces

• Coordination models for mobility
What is a coordination model

Historically, Linda was introduced as a new model for parallel programming, more flexible and high level wrt 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

It opened the way for a new research area: Coordination Languages and Models

A coordination model is an abstract (semantic) framework useful to study and understand problems in designing concurrent and distributed programs

“A coordination model is the glue that binds separate activities into an ensemble” [CarGe92]

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 agent creation and destruction, control of communication flows among agents, control of spatial distribution and mobility of agents, as well as synchronization 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 architectures

Examples of coordination architectures are

- the client-server
- the master-worker
- the software pipeline
- the blackboard
- the shared repository

We say that these are architectures because their software components are arranged in some special, well-defined (reusable) structures that enact a specific coordinated behavior.

Currently, software designers usually design a distributed software architecture using low-level communication primitives and/or special module configuration languages.

Instead, we suggest that a software designer should have clear a coordination model, embedded in a coordination language, able to support a specific coordination architecture.
Coordination models and languages

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)

Definition:
“A coordination language is the linguistic embodiment of a coordination model” and consists of some coordination mechanisms that are added to a host (sequential) language.

A coordination language should offer specific tools for optimizing coordination programs and reasoning on them.
Coordination languages

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.

- **C+ (original) Unix**
  - coordinable entities: processes
  - coordination media: files, anonymous pipes, signals
  - coordination laws: those enforced by Unix sys calls

- **Linda ("shared tuple space")**
  - coordinable entities: active tuples
  - coordination media: tuple space
  - coordination laws:
    - non blocking `out` of passive tuples;
    - blocking `read/in` by pattern matching;
    - complex `eval` semantics

- **GAMMA**
  - coordinable entities:
    "chemical reaction" represented as condition-action pair
  - coordination media: multiset
  - coordination laws: fixpoint application of condition-action pairs

- **PoliS**
  - coordinable entities: tuple spaces
  - coordination media: “root” tuple space
  - coordination laws: Linda + inter-tuple space laws

- **Interaction Abstract Machines**
  - coordinable entities: objects as multisets
  - coordination medium: forum
  - coordination laws: a fragment of Linear Logic + broadcasting
Linda formal semantics

Linda was defined at Yale and implemented over several 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 90]

[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.
A formal specification of Linda

The following transition system specifies Linda coordination model

**Coordination entities:**

Type names: \( \text{Type} = \{\text{int}, \text{char}, \ldots\} \)

Typed values: \( \text{Value} = \bigcup \{a: \tau, \perp: \tau \mid a \in \forall \tau\} \)

Passive tuples: \( \text{Tuple} = \text{Value}^i \)

Active tuples: \( \text{Active} = (\text{Value} \cup \text{Process})^i \)

Linda ops: \( \text{Op} = \{\text{eval}(t) \mid t \in \text{Active}\} \cup \{\text{out}(s), \text{rd}(s), \text{in}(s) \mid s \in \text{tuple}\} \)

Linda processes: \( \text{Process ::= } \Gamma p, \)

Tuple space: \( \text{TS} = \bigoplus \{t: \#[t]\} \)

**Coordination medium:**

Tuple space:

\( \text{Linda } \Gamma, \rightarrow \text{ where } \Gamma = \text{TS } \text{ and } \text{TS } \times \text{TS} \)

**Coordination rules:**

Process creation: \( \forall t \in \text{Active} : \{t'[i: \text{eval}(t).e]\} \rightarrow \{t'[i:e], t\} \)

Tuple creation: \( \forall t \in \text{Tuple} : \{t'[i: \text{out}(t).p]\} \rightarrow \{t'[i:p], t\} \)

Tuple copying: \( \forall s, t \in \text{match} : \{t'[i: \text{rd}(s).p], t\} \rightarrow \{t'[i:t.p], t\} \)

Tuple removal: \( \forall s, t \in \text{match} : \{t'[i: \text{in}(s).p], t\} \rightarrow \{t'[i:t.p]\} \)

Local transition: \( p' \rightarrow p'' \)

\( \{t'[i:p']\} \rightarrow \{t''[i:p'']\} \)

\( ts' \rightarrow ts'' \)

\( ts \oplus ts' \rightarrow ts \oplus ts'' \)
The Linda calculus

The transition system Linda is defined independently of the host language and any implementation concern: it is a specification of the coordination model underlying Linda.

In order to have a framework useful for abstract implementations, we have also introduced a Linda calculus

\[ P ::=} \text{eval}(P).P \mid \text{out}(t).P \mid \text{rd}(t).P \mid \text{in}(t).P \mid X \mid \text{recX}.P \mid P[] \mid \text{end} \]

Interleaving transition system SOS style

\[
\begin{align*}
\text{eval)} & \quad M \oplus \text{eval}(P).P' \rightarrow M \oplus P \oplus P' \\
\text{out)} & \quad M \oplus \text{out}(t).P' \rightarrow M \oplus P \oplus t \\
\text{rd)} & \quad M \oplus \text{rd}(s).P' \oplus t \rightarrow M \oplus P [t/s] \oplus t, \text{if} \ (s,t) \in \text{match} \\
\text{rd)} & \quad M \oplus \text{in}(s).P' \oplus t \rightarrow M \oplus P [t/s], \quad \text{if} \ (s,t) \in \text{match} \\
\text{rec)} & \quad M \oplus P[\text{recX}.P/X] \rightarrow M \oplus P' \\
& \quad M \oplus \text{recX}.P \rightarrow M \oplus P' \\
\text{leftchoice)} & \quad M \oplus P[]P \rightarrow M \oplus P \\
\text{rightchoice)} & \quad M \oplus P[]P \rightarrow M \oplus P' \\
\text{end)} & \quad M \oplus \text{end} \rightarrow M
\end{align*}
\]

Multistep (parallel) transition system SOS style

\[
\begin{align*}
\text{par)} & \quad M1 \rightarrow M1' \quad M2 \rightarrow M2' \\
& \quad M1 \oplus M2 \rightarrow M1' \oplus M2'
\end{align*}
\]
Choices in designing a semantics for a Linda-like language

[Campbell et al. 96] have studied and compared a number of formal semantics for Linda, and found a number of options in describing its coordination model

• simple matching / constrained matching?
• eval is the only means of process creation?
• active tuples actually exist?
• active fields in an active tuple are executed in parallel?
• active tuples are first class objects?
• bulk retrieval operations, if possible, are atomic?
• tuple space name always specified?
• tuple spaces are first class objects?
• scoping hierarchy of tuple spaces?
• garbage collection of tuple spaces?
• a global (root) tuple space as operating environment?
Coordination based on multiset rewriting

Conventional languages for distributed programming are based on primitives like send/receive which

- distinguish between process, data, msg, and channel
- require a synchronous style to simplify programs
- require an id: correspondent name, mailbox, channel

A multiset is a versatile data structure which can be used:

• to represent and handle a process (object) state
• to represent and handle a channel state
• to represent and handle the state of a set of threads

Whereas even Petri Nets can be represented as multiset rewriting systems, a computing model called CHAM (chemical abstract machine) [Berry&Boudol 93] best represents non sequential computations based on multiset rewritings

Linda can be semantically mapped on the CHAM, and most coordination languages can be easily mapped on it (you need a coordination language to implement a coordination model!!)
The Chemical Abstract Machine

A tuple space is actually a multiset of tuples; abstractly, a multiset is also called *bag*.

We could imagine an observer that sees what happens in a tuple space: he could describe what he sees as *a chemical soup* (namely a solution) which is *active*, namely molecules in the solution appear and disappear.

A Chemical Abstract Machine [Berry Boudol 93] is a triple \((G, C, R)\) where

- \(G\) is the grammar generating \(C\)
- \(C\) is the set of configurations (molecules)
- \(R\) is a set of rules: \(condition \times bag(C) \times bag(C)\)

where *condition* is a logical expression on molecules in the bags, which respectively have to be deleted (preconfiguration) and added (postconfiguration).

Rules can fire concurrently if they do not interfere; if some rules conflict on the same molecules, a non conflicting subset is chosen non deterministically to react.

Molecules are defined as terms of a specific algebra; solutions are finite multisets of molecules; a solution can be enclosed in a membrane, denoting a separate subsolution.
General laws of any CHAM

- **Reaction** law: an instance of right-hand side of a rule can replace the corresponding instance of its left-hand side.

Given the rule \( L_1, L_2, \ldots, L_k \rightarrow R_1', R_2', \ldots, R_j' \) if \( L_1, L_2, \ldots, L_k \) matches the solution \( M_1, \ldots, M_s \) then we apply the rule and transform the solution

- **Chemical** law: reactions can be performed freely within any solution (i.e. when a subsolution evolves its supersolution is also considered to have evolved)

\[
\frac{S \rightarrow S'}{S \oplus S'' \rightarrow S' \oplus S''}
\]

- **Membrane** law: a subsolution \( S \) can evolve freely within any context \( C[.] \)

\[
\frac{S \rightarrow S'}{[C[S]] \rightarrow [C[S']]}\]

- **Airlock** law: a molecule \( m \) can always be extracted from and reabsorbed into a solution \( S \), insofar as its identity as an individual molecule is preserved

\[
m \oplus S \rightarrow m \triangleleft [S]
\]
A CHAM for Linda

We start from a toy Linda formalism called Linda calculus.

\[ P ::= \text{eval}(P).P \mid \text{out}(t).P \mid \text{rd}(t).P \mid \text{in}(t).P \mid X \mid \text{rec}X.P \mid P[P] \mid \text{end} \]

The three new productions are useful to model recursive processes and local nondeterministic choice.

- \text{par}) \quad M1 \oplus M2 \leftrightarrow M1, M2
- \text{eval}) \quad \text{eval}(P).P' \rightarrow P,P'
- \text{out}) \quad \text{out}(t).P \rightarrow P,t
- \text{rd}) \quad \text{rd}(s).P,t \rightarrow P[t/s],t \quad \text{if} \ (s,t) \ \text{match}
- \text{in}) \quad \text{in}(s).P,t \rightarrow P[t/s] \quad \text{if} \ (s,t) \ \text{match}
- \text{rec}) \quad \text{rec}X.P \rightarrow P[\text{rec}X.P/X]
- \text{leftch}) \quad P[.] P' \rightarrow P
- \text{rightch}) \quad P[.] P' \rightarrow P'
- \text{end}) \quad \text{end} \rightarrow

We have also defined pattern matching in CHAM [CJY94].

This semantics is simple and fully distributed; it can be used to study other coordination languages and novel mechanisms.
Implementing the CHAM

The CHAM itself has been suggested as coordination language for a MIMD architecture [Ma et al., 1996 - Australia]

They see the CHAM as a refinement of Gamma (see below)

\% parallel summation
initialization
   m = 10, m= 20, m = 15
reaction rules
   x,y leadsto z by \{z = x+y\}

\% sleeping barber problem
initialization
   [i:1..N]:chair=TRUE; bsp=TRUE
reaction rules
   pin,bsp leadsoto pcut,bwk;
   pin,chair leadsto pwt when (~bsp)
   pcut, bwk leadsto pout, bfin;
   pwt, bfin leadsto pcut,chair,bwk;
   bfin leadsto bsp when (|pwt|==0)

This CHAM-based language has been implemented over an AP1000, with 128 nodes.
Designing a distributed runtime using the CHAM

The CHAM model has been used to describe and study the runtime systems of coordination languages:

- **Linear Objects** [Andreoli et al 1993]
  The main issue is the description of broadcasting communication typical of LO

- **Facile** [Leth and Thomsen 1994]
  The main issue is the description of a distributed execution management system necessary to implement Facile over a network

- **Java** [Ciancarini and Laneve 97]
  We have studied the run time behavior of concurrent/distributed Java programs

The CHAM has a linguistic counterpart in GAMMA
GAMMA

GAMMA (General Abstract Model for Multiset mAnipulation) [Banatre & LeMetayer 90] is a coordination model whose main data structure is the multiset (or bag) and whose unique control structure is the $\Gamma$ operator

\[
\Gamma((R_1,A_1),\ldots,(R_m,A_m))(M) = \\
\text{if } \forall i \in [1,m], \forall x_1,\ldots,x_n \in M, \sim R_i(x_1,\ldots,x_n) \text{ then } M \\
\text{else let } x_1,\ldots,x_n \in M, \text{ let } i \in [1,m] \text{ such that } R_i(x_1,\ldots,x_n) \text{ in } \\
\Gamma((R_1,A_1),\ldots,(R_m,A_m))((M\setminus\{x_1,\ldots,x_n\}) + A_i(x_1,\ldots,x_n))
\]

The notation \{…\} represents multisets; $(R_i,A_i)$ are pairs of closed functions (no global variables) specifying reactions. The effect of $(R_i,A_i)$ on multiset $M$ is to replace in $M$ a subset of elements $\{x_1,\ldots,x_n\}$ such that $R_i(x_1,\ldots,x_n)$ is true by the elements of $A_i(x_1,\ldots,x_n)$

$\Gamma$ is a fixpoint operator: reactions continue until no new reaction is possible (implicit termination condition).

All possible reactions are fired: this is the source of parallelism in GAMMA.

**Examples:**

- sum_all: $x,y \rightarrow x+y \Leftarrow true$
- max: $x,y \rightarrow y \Leftarrow x \leq y$
- sort: $(i,v),(j,w) \rightarrow (i,w),(j,v) \Leftarrow (i>j) \land (v<w)$
Programming by rewriting a multiset

The sieve of Eratosthenes can be written as follows:

sieve: x,y \rightarrow y \leftarrow \text{multiple}(x,y)

sieve({2,3,4,5,6,7,8}):
GAMMA Calculus

GAMMA has been enriched with a calculus [Hankin & al] including some program composition operators:

\( P_1 \odot P_2 \) (sequential) and \( P_1 + P_2 \) (parallel)

**Example1:**
- \( \Gamma_1: \text{n} \rightarrow \text{n-1}, \text{n-2} \Leftarrow \text{n}>1 \)
- \( \Gamma_2: \text{n} \rightarrow 1 \Leftarrow \text{n}=0 \)
- \( \Gamma_3: \text{n},\text{m} \rightarrow \text{n+m} \Leftarrow \text{true} \)

\( \Gamma_3 \odot \Gamma_2 \odot \Gamma_1 \) computes the \( n \)-th Fibonacci number;
we can prove: \( \Gamma_3 \odot \Gamma_2 \odot \Gamma_1 = \Gamma_3 \odot (\Gamma_2 + \Gamma_1) \)

However, the original GAMMA calculus presents some semantics problems, that have been studied and solved in [Ciancarini et al. 95]

**Example2:**

\( \text{sort: match} \odot \text{init} \)

where \( \text{init:} \ (x \rightarrow (1,x) \Leftarrow \text{integer(x)}) \)

\( \text{match:} \ ((i,x),(j,y) \rightarrow (i,x),(i+1,y) \Leftarrow (x\leq y \text{ and } i=j)) \)

To have the equivalence \( \text{match} \odot \text{init} = \text{match} + \text{init} \)
a synchronized termination of \( \text{match} \) and \( \text{init} \) in parallel combination is required
A list of coordination models

Classic OS theory:
• Shared Data / semaphores and monitors
• Message passing / remote procedure calls

AI coordination architectures
• Blackboard [Nii, Bisiani & Forin]
• KNOS [Tsichritzis & Fiume & Gibbs & Nierstrasz]

Models for non imperative languages
• The Kahn-McQueen model
• Actors [Hewitt & Agha]
• Unity/PCN language model [ManyChandy et al.]
• Concurrent constraint logic prog. [Shapiro, Saraswat]

Event-driven coordination languages
• Manifold [Arbab]

Coordination models based on multiset rewriting
• Gamma (and CHAM) [Banatre & LeMetayer; Berry&Boudol]
• Multiple tuple spaces [Gelernter, Hupfer]
  Bauhaus [Carriero & Gelernter & Hupfer]
  PoliS [Ciancarini et al.]
Interaction Abstract Machines [Andreoli & Pareschi]
Blackboard

The *blackboard* architecture [Nii] was introduced to design some AI applications in which
- a number of coordination entities called “*knowledge sources*” cooperate to solve a problem
- described by data contained in a coordination medium called the “*blackboard*”, controlled by an *agenda* module;
- the control module in early applications was sequential, but it is possible to distribute both the blackboard and the activities included in the agenda

A physically distributed logically shared data structure is abstractly 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
KNOs (KNowledge acquisition, dissemination, and manipulation Objects) [Tsichritzis et al. 1987] is a coordination model introduced for knowledge based systems with multiple agents that can negotiate, cooperate and learn.

A set of cooperating KNOs exist within a context (typically, a context is associated to a workstation); KNOs communicate reading or writing message on a blackboard. A KNO can move to another context (migration); it can modify itself accepting code for new operations (learning).

KNOs behaviors can be written using a LISP-like notation:

```
(rule <rule-name> (trigger <trigger-condition>)
  (action <action-series>))
```

Actions are of five different types:
- local actions (on local variables)
- communications (using the blackboard)
- spawn, die, move, freeze, unfreeze a KNO
- learning actions (send or receive new behavior rules)
- limb actions (grow, kill, ship, teach, unteach a limb)

A limb is a KNO generated by another KNO (head). A head and its limbs can span different contexts.
Manifold

Manifold [Arbab 91] offers a graphic formalism to coordinate activities

In *Manifold*, the basic components are processes, events, ports and streams; a process is a blackbox with ports

```plaintext
example()
port in input
port out output
{ process A is A_type.
  process B is B_type.
  process C is C_type.
  start: (activate A, activate B, activate C); do begin.
  begin: (A → B, output → C, input → output).
  e1: (B→input, C→A, A→B, output→A, B→C, input→output)
  e2: C → B.
}
```

**Starting configuration**  **Re-Configuration after event e2**
Multiple Tuple Spaces

C-Linda was designed for simplifying programming on different parallel architectures, supporting a high degree of portability; however, C-Linda offers good support also for system programming of distributed systems.

“Normal” Linda applications: programs explicitly exploiting massive parallelism, typically with a master/worker architecture (“closed” Linda programs).

A single Tuple Space has the problem that it has to be “closed”, i.e. it must be used by a single parallel computation (otherwise processes of different computations could interfere with each other);

An obvious extension to the Linda model is based on Multiple Tuple Spaces.

The multiple tuple spaces model is especially useful for “open” Linda programs, in which several users interact with a Linda program that coordinates them.

Multiple tuple spaces [Gelernter 89] can offer a natural mechanism of modularity; the concept is also useful for multiparadigm systems; persistent tuple spaces can be defined in this framework.

There are several proposals for MTS models:
- Paradise and PoliS
- Interaction Abstract Machines and CLF
- Bauhaus
- LIME
Paradise

Paradise implements a simple “flat” multiple tuple space concept: the tuple spaces are named and they cannot be nested.

Tuple spaces are protected under an access control scheme.

Tuple spaces can be used as shared relational databases, and transactions can be defined.

A Paradise client can use tuple spaces under control of different TS servers.

There are several ways that programs may share data using Paradise:

- one program may visualize or analyze data generated by other programs
- a program may compare and cross-analyze data generated by other programs
- a program may use the data from another program to perform its own computations or simulations
PoliS

PoliS [Ciancarini et. al. 91, 97] tries to be more faithful to Linda than Paradise: it adds a new operation to create a named “empty” tuple space:

\[ tsc(tuple\_space); \]

PoliS also extends out operations to deal with remote creation of tuples (eg.: \[ out(tuple)@tuple\_space; \])

A “meta” tuple space takes care of these messages, so that generative communication is used for interspace coordination as well

A closed PoliSpace is similar to a Linda program: a number of tuple spaces start and run until completion of the global task

"Closed" PoliS programs are intended for massive computations

An open PoliSpace, i.e. an interactive distributed program, includes a number of tuple spaces and some spaces called “shells” (or “roles”): a shell is an “interactive” tuple space, where the user can directly put tuples (a shell is an interaction point between the PoliSpace and the external environment)

An open PoliSpace is a useful model for groupware applications, e.g.:
- a distributed multiuser sw dev. environment,
- an electronic mail system,
- a financial simulation of stock exchanges,
- a Web-based card playing system
PoliS

A PoliSpace can be seen as a “city”, i.e. a set of distinct spaces; each space contains several agents that
- can output objects to any space they know;
- can only look in their own space for input;
- interspace communication mechanism is generative communication through a meta tuple space

PoliS is a coordination model: to obtain a coordination language, it has to be combined with a sequential language;

The combination must specify:
- type system for tuples and matching
- name system for tuple spaces
- semantics of agent (eval) and tuple space (tsc) creation
- control of tuple space operations
- modularity mechanisms for programs
- communication mechanisms
- termination mechanisms for tuple spaces

In roder to reason on PoliS specification documents, we have developed
- a TLA based logic and theorem prover
- a CTL-based logic and model checker
Interaction Abstract Machines

The model of Interaction Abstract Machines (IAM) [ACP 93] has been introduced to describe interactions among independent, locally defined subsystems.

IAMs combine dual concepts in distributed problem solving, such as blackboards broadcast communication, which are exploited to account for, respectively, the tight integration and the loose integration of system components.

In the IAM metaphor, the state of a (single) agent is represented as a set of particles evolving within a determined (computational) space.

Each particle represents a resource, and we assume that the particles are in a perpetual "Brownian" motion, so that they may randomly collide together.
Single agent IAM

In a single-agent IAM, the state of the agent is represented as a multiset of resources. It may evolve according to rules which rewrite the multiset.

In fact, a single-agent IAM can be seen as an elementary blackboard system:

- the state of the agent captures the content of the blackboard at any time;
- the rules (or methods) represent the different knowledge sources which interact through the blackboard:
  - the head of a rule specifies resources which are taken from the blackboard;
  - the body specifies resources which are output to the blackboard.
Multi-agent IAM

To build multi-agent systems, we add new primitives for the creation and termination of agents.

The body of a method may now be formed of 0 or more multisets of resources instead of just one unique multiset.

The generated multisets become completely independent agents and start evolving autonomously.

The universe (of computation) now consists of several concurrent spaces of the kind introduced previously.
Linear Objects

Linear Objects (LO) [Andreoli Pareschi 91] is an object based language based on the IAM coordination model.

An LO program is a set of methods (rewriting rules)
An agent is represented by a multiset
Interagent coordination is possible via broadcast

\[ \text{<multiset> <broadcast> <built-ins> o- <goal>} \]

**Multisets** are described as: \( a_1 @ \ldots @ a_n \)

**Goals** have the forms: \( a_1 @ \ldots @ a_1 | \text{goal}_1 & \ldots & \text{goal}_n | T \)

A **query** is \(<P, G>\) where \(P\) is a program and \(G\) a goal

**Semantics:**
Given a query \(<P,G>\), a computation consists of the construction of a **target_proof**, that is a tree whose nodes are labelled by LO sequents of the form \(P |- C\); branches are obtained using the following inference rules:

- **Decomposition**
  
  \[
  \begin{align*}
  P &\vdash G_1, G_2, C & P &\vdash G_1, C & P &\vdash G_2, C \\
  P &\vdash G_1 @ G_2, C & P &\vdash T, C & P &\vdash G_1 & G_2, C
  \end{align*}
  \]

- **Propagation:** if \((\text{Head} @ \text{Tell o- G}) \in P\)
  
  \[
  P &\vdash G_1, C \\
  P &\vdash || \text{Head} @ \text{Tell} ||, C
  \]

A **target_proof** for a query \(<P,G>\) is a tree whose root is \(P |- G, C\) Leaves are labelled by \([T]\)
A proof tree

Suppose that the method $p \ @ q \ o\rightarrow (r@s) \ & t$ is triggered in the multiset $\{p, q, u\}$

The resulting computation is depicted by the proof tree

$$
\frac{r, s, u}{r@s, u} \quad \frac{t, u}{(r@s) & t, u} \quad \frac{(r@s) & t, u}{p, q, u}
$$

In other words, the computational interpretation of linear sequents can mirror the IAM
Bauhaus

Bauhaus [Gelernter et al 95, Hupfer 96] is a generative, uncoupled, associative memory coordination model, like Linda.

However:

• Bauhaus is based on (nested) multisets
• matching is based on set inclusion
• primitives return multisets

Example: a multiset in Bauhaus
\[ M = \{a, b, \{b\}, X, \{c, \{d, e\}\} \} \]

Bauhaus generalizes Linda in the following ways:
- tuples and tuple spaces are both replaced by msets
- tuples and tuple templates disappear
- passive and active data objects are unified

One multiset matches another if it is included in the other

Bauhaus has been explored in [Hupfer 1996], and has been used to implement a number of groupware applications
All Bauhaus coordination programs are evaluated wrt a universal coordination space (ucs) which is the basic mset

Example 1 (in). Suppose that the current ucs is

\{a, b, \{b\}, X, \{c, \{d, e\}\}, \{c, Y\}, Z\}

where X, Y, Z represent processes (live msets)

Let X be a process executing the following code:

```mset m;...; i=in({c});```

There are two possible results:

- \{a, b, \{b\}, X, \{c, Y\}, Z\}; \ i=\{c, \{d, e\}\}
- \{a, b, \{b\}, X, \{c, \{d, e\}\}, Z\}; \ i=\{c, Y\}

Example 2 (out). Suppose that the current ucs is

\{b, \{b\}, X, \{c, \{d, e\}\}, Z\}

Let X be a process executing the following code:

```mset m;...; i=in({v, \{w\}});```

The result is:

\{a, b, \{b\}, X, \{c, Y\}, Z, \{v, \{w\}\}\};
Bauhaus operations: move

Bauhaus adds to Linda’s classic operations \texttt{in/read/out} (extended to deal with multisets) the \texttt{move} operation, which is in some way the opposite of \texttt{in}: a process moves to a matching multiset (if available; otherways it blocks), so implementing a form of associative migration.

Using \texttt{move}, processes can move up or down in the ucs one level at a time.

Example 3 (move). Suppose that the current ucs is

\begin{verbatim}
{ a, b, {b}, X, {c, Y}, Z, {v, {w}} }
\end{verbatim}

Let X execute the following code: \texttt{move({v})};

The result is:

\begin{verbatim}
{ a, b, {b}, {c, Y}, Z, {v, {w}, X} }
\end{verbatim}

It is also available the non blocking form \texttt{movep}.

The hierarchical and persistent nature of multisets makes them an attractive means of storing Web documents.

Both a Web server and browser have been implemented emulating Bauhaus coordination using Netscape and CGI.

Each page offers 4 navigational commands: up, top, back, down.

The pages are active, that means for instance that agents which are visiting them are visible by other agents.
A conversation stream system

As an example of Bauhaus coordination design, we describe a multiuser conference system implemented in C-Bauhaus and Tcl/Tk.

Software architecture of conversation stream system
Coordination and I/O

Agent Agents are active, self-contained entities performing actions on their own behalf.

Action Actions can be divided into two different classes:

- Inter-Agent actions: these actions involve different agents. They are the subject of coordination models.

- Intra-Agent actions: these actions are performed by a single agent to perform computation as well as all communication outside the coordination model, like primitive I/O operations or interactions with users.

Coordination consists of managing the inter-agent activities of agents collected in a configuration.

Configuration a \textit{structured} collection of interacting agents

Remark In general, one could argue that I/O with the operating environment can easily be subsumed by a coordination model; however, this is true only to some extent.

Real-world applications have to fit into environments with agent-like user interfaces and I/O devices which do not adhere to any clearly defined coordination model.

In order to integrate such interaction into a system of coordinated agents in a clean way, it is preferable to introduce specialized agents which play the role of interfaces between the world of coordinated agents and the outside environment.
Internet programs are distributed and mobile

New computing paradigms based on code distribution and mobility need novel specification and programming languages.

A key issue when an application includes mobile components is how to design its architecture, which can be decomposed in at least three different layers:

• The physical network layer, made mostly of immobile hosts and reliable and fast connections, where a mobile entity consists of a piece of hardware using a connection usually unstable (e.g. wireless) and with low bandwidth.

• The middleware network layer, made of abstract machines (e.g. a JavaVM, an XWindow server, etc.), where a mobile entity consists of a whole process or a service migrating from a host to another host. Problems at this level are resource management (e.g. load balancing) and service reliability (e.g. security).

• The logical network layer, made of application code scattered over the middleware network, where a mobile entity consists of an agent migrating from an abstract machine to another one. A problem at this level is how such a mobile entity can cooperate with other entities in the application.
Kinds of Mobile Entities

 Mobility of data
 Clients and servers usually only exchange data

 Mobility of reference
 Each agent has a reference to a “working location”,
 but it can change such a reference, “moving” by changing
 its bindings inside a static structure

 Code mobility [plain Java]
 Programs move (on demand) from a site to another

 Agent migration [Aglets, etc]
 Agent images (code & store) can move from a
 location to another, travelling some “itinerary”

 Threads migration
 Agent images and scheduling state (code & store &
 execution state) travel some “itinerary”

 Closure mobility [Cardelli’s Obliq]
 An agent moves and keeps its environment

 Ambient mobility [Bauhaus, Ambient, Lime]
 Both agents and their operating environments can
 move
Mobility of data and references

The original WWW was based on two concepts of mobility:

⇒ HTTP servers distribute on request HTML pages to client browsers: this is *mobility of data* (HTML is not Turing equivalent: it is just a SGML dialect)

⇒ A browser can “navigate” through the hypertext links, requesting HTML pages to a server, and sometimes “jumping” from a server to another server. Although URLs denote static, “physical” resources (URIs would be better!), they can be passed around: this is *mobility of references*

**Remark:** Mobility of reference means both that a channel name can be passed around, and that a process can detach a channel and connect another channel
# Coordination and mobility

The issue of coordination mechanisms for mobile agents has been studied especially in the context of environments for network-aware programming.

Multiple Tuple Spaces are natural coordination media for both mobile code (Java) and mobile agents (code+state).

The following proposals differ in the coordinable entities, in the mechanisms used to access the tuple spaces, and in the possibility of extending the coordination laws.

Details are discussed in [COZ99]

<table>
<thead>
<tr>
<th>System</th>
<th>Coordinables</th>
<th>Media</th>
<th>Language</th>
<th>Laws</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaSpaces</td>
<td>JavaAgents</td>
<td>TS</td>
<td>Java+Linda</td>
<td>Fixed</td>
</tr>
<tr>
<td>Lime</td>
<td>MobileAgents</td>
<td>Local and global TS</td>
<td>Linda+asynch primitives</td>
<td>Fixed</td>
</tr>
<tr>
<td>MARS</td>
<td>Java Agents</td>
<td>Network aware TS</td>
<td>Java+Linda</td>
<td>Programmable (in Java)</td>
</tr>
<tr>
<td>PageSpace</td>
<td>Agents+Services</td>
<td>Network unaware TS</td>
<td>Java+Linda</td>
<td>Fixed</td>
</tr>
<tr>
<td>TSpaces</td>
<td>Agents</td>
<td>Network unaware TS</td>
<td>Linda+user-defined primitives</td>
<td>Programmable (Overriding)</td>
</tr>
<tr>
<td>TUCSON</td>
<td>Information agents</td>
<td>Network aware TS</td>
<td>Logic oriented Linda</td>
<td>Programmable (in Prolog)</td>
</tr>
<tr>
<td>WCL</td>
<td>Agents, applications</td>
<td>Automatic reallocation</td>
<td>Linda+asynch primitives</td>
<td>Fixed</td>
</tr>
</tbody>
</table>
Conclusions

Coordination models are a new exciting research field

Coordination models are an important tool for a new class of distributed applications, in which several agents (humans, tools, programs) have to be coordinated.

There are several questions that are being addressed:

⇒ which coordination mechanisms are more expressive and useful?

⇒ which semantic models should be used to study such mechanisms?

⇒ which implementation techniques are best?

⇒ which software architectures match some specific coordination requirements?

⇒ which programming logics can be used to reason about coordination programs?

⇒ which new applications can be developed, which exploit the new technology?