Different Facets of Coordination

George A. Papadopoulos

Department of Computer Science
University of Cyprus
george@cs.ucy.ac.cy
Roots of Coordination _ 1

_ Multilinguality is able to:
  – Support diverse programming paradigms
  – Provide interoperability between them
  – Accommodate diverse execution models
  – Combine code written in a mixture of them but also provide orthogonal programming interfaces

_ Typical cases of realizing a multilingual framework is by means of Module Interconnection Languages and Compiler Target Languages
Roots of Coordination _ 2

_ Multilinguality is closely related to heterogeneity, since heterogeneous systems demand that the language to be used must support different models of computation (difficult)

_ Thus, we resort to using a mixture of languages

_ Related historical models are also those of:
  – Blackboard systems, developed traditionally for DAI
  – Objected-Orientatation and Actor systems
The Coordination Paradigm

- Separates the computational concerns in some system from the other concerns
- “Computational” can mean a number of things:
  - Execution of software components
  - Operation of hardware devices
  - Behaviour of human beings
- “Other concerns” can also have different meanings such as communication, cooperation, synchronization, etc.
What is Coordination

- Coordination is managing dependencies between activities (Malone and Crowston)
- Coordination is the process of building programs by gluing together active pieces (Carriero and Gelernter)
- Coordination is the additional information processing performed when multiple, connected actors pursue goals that a single author pursuing the same goals would not perform
Coordination Models and Languages

- A *coordination model* is the glue that binds together active pieces
- Ciancarini defines it as a triple \(<E, L, M>\)
  - \(E\) are the entities being coordinated
  - \(L\) the media used to coordinate them
  - \(M\) the semantic framework, the mode adheres to

- A *coordination language* is the linguistic embodiment of a coordination model
Classification of Coordination Formalisms

_ How one should classify coordination models and languages?

– In terms of the nature of what is being coordinated (types of components)?
– In terms of the kind of languages being involved?
– In terms of application domains?
– In terms of underlying architectures assumed?
– In terms of other issues such as scalability, openness, etc?
Defining the State of Computation: Data- vs Control-Driven Coordination

In the data-driven category of models, the state of the computation is usually defined in terms of both what is being coordinated (i.e. the data being sent or received) and how coordination is achieved (i.e. the coordination patterns employed).

In the control-driven category, the state is usually defined in terms only of the configuration apparatus set up between the involved components, data itself is of little significance.
Bird-Eye’s View of Data-Driven Coordination

- A process is interested in both handling data as well as setting up coordination patterns
- Stylistically and linguistically, coordination code is intermixed with computation code
- Usually, the coordination “language” is a set of primitives that have to use a host programming language
- The communication medium is essentially based on the Virtual Shared Memory metaphor
Bird-Eye’s View of Control-Driven Coordination

- There is a clear separation of components handling data from those that set up coordination patterns
- The two types of code are also clearly separated
- A fully-fledged coordination language is required to work together with some computational one(s)
- Point-to-point “Occam” type of communication is employed with processes having well defined input-output interfaces
Data-Driven Coordination
Formalisms
Main Characteristics of Data-Driven Coordination Formalisms

- VSM is realized as a shared dataspace, a common, content-addressable data structure
- It is independent in time and space
- Data is represented as a generic tuple structure, in some cases flat, in other cases structured
- Mechanisms for retrieving data vary, and include pattern matching, multiset rewriting, etc.
- Issues of security and openness are handled by adopting a non-flat structure with localized access
Linda

- A set of 4++ primitives (in and inp, rd and rdp, out, eval) are used to access the **Tuple Space** by associative *pattern matching*

- Easy to use, less easy to implement efficiently, issues of tuple storage, exact implementation of eval, etc.

- Linda has many friends: C, Prolog, Java, Eiffel, to name but a few; also, it can coexist nicely with many paradigms (imperative, declarative, o-o)
Dining Philosophers in Linda

```c
#define NUM 5

philosopher(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");
    }
}

main()
{
    int i;
    for (i=0, i<=NUM, i++)
    {
        out("fork",i);
        eval(philosopher(i));
        out("room ticket");
    }

    out("fork",i);
    out("room ticket");
}
```
Piranha: Better Load Balancing for Linda _ 1

_ Features *adaptive parallelism*, i.e. processor assignment to processes changes dynamically
_ A feeder distributes computation and collects results, a number of piranhas perform computations
_ Piranhas are statically distributed over a network of w/s, and don’t migrate
_ They are active if the workload of the node on which they reside allows it
Piranha: Better Load Balancing for Linda _ 2

Piranhas retreat when the node on which they reside is claimed back by the system, post the rest of the work to the tuple space for other piranhas.

Typical application areas of the model is scientific computing (LU decompositions and Monte Carlo simulations).
Typical Structure of a Piranha program

```c
#define DONE -999
int index;

feeder()
{
    int count;
    struct Result result;

    /* put out the tasks */
    for (count=0; count<TASKS; count++)
        out("task", count);

    /* help compute results */
    piranha();

    /* collect results */
    for (count=0; count<TASKS; count++)
        in("result",count,result_data);
}

piranha()
{
    struct Result result;

    while (1)
    {
        in("task",index);
        if (index==DONE)
        {
            /* all tasks are done */
            out("task",index);
        }
        else
        {
            /* do the task */
            do_work(index,&result);
            out("result",index,result);
            in("tasks done",?index);
            out("tasks done",i+1);
            if ((i+1)==TASKS)
                out("task",DONE);
        }
    }
}
```

retreat()
{
    /* replace current task */
    out("task",index);
}
Bonita: Finer notion of Tuple Retrieval _ 1

_ Effectively differentiates in the in and rd operations between asking for a tuple and actually getting it

_ rquid=dispatch(ts,tuple)
  puts tuple in ts and returns a tuple id to be used by other processes to retrieve it

_ rquid=dispatch(ts,template,d|p)
  retrieves a tuple from ts matching template, by removal (d) or copying (p), and returns its id
Bonita: Finer notion of Tuple Retrieval _ 2

_ rquid=dispatch_bulk(ts1,ts2,template,d|p)
  moves (d) or copies (p) from ts1 to ts2 all tuples matching template

_ arrived(rquid)
  moves (d) or copies (p) from ts1 to ts2 all tuples matching template

_ obtain(rquid)
  suspends until the tuple rquid is available
Retrieving Tuples in Bonita

Linda

Bonita

int rqid1, rqid2, rqid3;

in("ONE");    rqid1=dispatch(ts,"ONE",d);
in("TWO");    rqid2=dispatch(ts,"TWO",d);
in("THREE");  rqid3=dispatch(ts,"THREE",d);

obtain(rqid1);
obtain(rqid2);
obtain(rqid3);

All dispatches are done in parallel, retrieving of tuples is overlapped
Finer Notion of Non-Deterministic Retrieval in Bonita

Linda

Bonita

int rqid1, rqid2;

rqid1=dispatch(ts,"ONE",d);
rqid2=dispatch(ts,"TWO",d);

while(1)
{
    if (inp("ONE"))
        {do_first(); break;}
    if (inp("TWO"))
        {do_second(); break;}
}

while(1)
{
    if (arrived(rqid1))
        {do_first(rqid1); break;}
    if (arrived(rqid2))
        {do_second(rqid2); break;}
}
Bauhaus Linda: More sophisticated Tuple Matching Based on Multisets

- No differentiation between active and passive tuples, tuples and tuple spaces, etc.

- \text{out } \{x->R\} \text{ applied to } \{a \{x \ y \ Q\} \{\{z\}\} \ P\} \text{ by } P \text{ yields } \{a \{x \ y \ Q \ R\} \{\{z\}\} \ P\}

- \text{mset } m:=\text{rd } \{x\} \text{ applied to } \{a \ b \ b \{x \ y\} \{\{z\}\} \ P\} \text{ by } P \text{ makes } m \text{ get the value } \{x \ y\}

- \text{mset } m:=\text{in } \{x\} \text{ applied to } \{a \{x \ y \ Q\} \{R \{z\}\} \ P\} \text{ by } P \text{ makes } m \text{ get the value } \{x \ y \ Q\} \text{ and becomes a live set (due to } Q)
Bauhaus Linda: Structured Multisets

- move \{w\} executed on
  \{a b b \{x y Q\} \{w \{z\}\} P\} by P, yields
  \{a b b \{x y Q\} \{w \{z\} P\}\}

- Two variants of move, up and down, cause the issuing process to move up and down the hierarchy
Objective Linda: “Object-Oriented Bauhaus Linda” for Open Systems

- Tuples and tuple spaces are substituted by objects and object spaces, the former described in an Object Interchange Language
- Object spaces are accessible through logicals, i.e. object space references
- Object spaces can be organized in hierarchies and communication can be achieved via several object spaces
Objective Linda’s Primitives _ 1

_ bool out(MULTISET *m, double tout)
  moves the objects in m into the object space, operation must be completed within tout secs

_ bool eval(MULTISET *m, double tout)
  similar, but the objects are also activated

_ bool in(OIL_OBJECT *o, int min, int max, double tout)
  tries to remove at least min but not more than max objects matching o within tout secs
Objective Linda’s Primitives _ 2

bool rd(OIL_OBJECT *o, int min, int max, double tout)
same as before, but copies of these objects are cloned

infinite_matches and infinite_time are used to indicate infinite number of matched objects in min or max, and no timeout constraints in tout respectively
Law-Governed Linda: Logical Regulation of Tuple Space Traffic

Whereas the previous models extend the basic language, Law-Governed Linda introduces controllers, which interpose themselves between the Tuple Space and the processes accessing it.

All controllers execute the “law” and allow traffic between every process and the tuple space only if it adheres to it.

A law typically specifies access rights, creates local spaces, enforces security mechanisms, etc.
A Secured Message Exchange in Law-Governed Linda

out([msg, from(Self), to(_)|_]) :- do(complete).
in([msg, from(...), to(Self)|_]) :- do(complete) :: do(return).
out([X|_]) :- not(X=msg), do(complete).
in/rd([X|_]) :- not(X=msg), do(complete) :: do(return).

where a message is of the form
[msg, from(s), to(t), contents]
LAURA: Linda for Modelling Open Distributed Systems

In LAURA, the common communication medium is a service space where agents post and retrieve offered services.

These are described as interface signatures comprising a set of operation signatures.

Communications among agents is realized by exchanging forms of three types: service-offer, service-request, and result-form.

For signatures a Service Type Language is used.
A Service Offered in LAURA

SERVE large-agency operation

(getflightticket : cc * <day,month,year> * dest
  -> ack * <dollar,cent>);

getbusticket : cc *
  <thedate.day, thedate.month, thedate.year> * dest
  -> ack * <dollar,cent> *line;

gettrainticket : cc * <day,month,year> * dest
  -> ack * <dollar,cent>).

SERVE

Three services are offered, the code of the selected service will be bound to operation
A Service Requested in LAURA

_SERVICE small-agency
(getflightticket: cc * <theday, thedate.month, thedate.year>
  * dest
  -> ack * <dollar,cent>).

SERVICE

 Small-agency invokes the specified service, passes along parameters such as cc and dest and waits for an ack message with the value of the ticket.
Ariadne/HOPLa: Linda for Collaborative Computing

- In Ariadne, the shared dataspace holds tree-shaped data, structured or semi-structured and type definitions governing its structure
- The associated Hybrid Office Language is used to model process behaviour in the form of flexible records
- Some useful constructors is Set for collections, Action for tasks to be performed either sequentially (Serie) or in parallel (Parl), etc.
Coordinating an Electronic Discussion in Ariadne/HOPLa

Discussion<Process(
    group -> Set+Action( type -> Actor;
                        value -> PS: set);
    discuss -> Thread<Data+Serie(
                        message -> String+Action(actor
                                             -> {p | p in PS});
                        replies -> Set+Parl(type -> Thread)))

First the group is defined, then the discussion starts with a triggering message, followed by replies in any order by the rest of the actors, each one of them starting an independent thread.
Sonia: Applying the Linda Metaphor to Modelling Activities in I.S.

_ Sonia is not really an extension of Linda, but rather an adaptation of the latter so that the Linda metaphor can be used in an intuitive way by non specialists in coordination or, indeed, C.S.

_ There is an agora, accessed by actors, the latter communicating by posting messages formed as named tuples

_ A timeout functionality is introduced, an integral element of any framework modelling I.S.
Communicating in Sonia

_ The usual out, in, and rd names have been replaced by more intuitive ones such as post, pick, and peek

_ There is also a cancel primitive for timeouts

_ Posted tuples are named such as Tuple(:shape "square" :color "red"), and are retrieved via templates such as Template(:shape any :color Rule("value='red' or value='blue'"))
Jada/SHADE: Linda for the WWW

_ The shared dataspace paradigm can be realized naturally and profitably in the WWW, and the coordination paradigm can be used for orchestrating the execution of web-based applications, such as groupware, workflow, etc.

_ Jada (Java - Linda) can be seen as a basic infrastructure for building such environments

_ It can be used for expressing mobile object coordination and multithreading (e.g. PageSpace)
Jada Model

- Classes such as TupleServer and TupleClient are provided for realizing remote access to a tuple space
- Communication is done via sockets
- A TupleClient must know the host and port-id of the TupleServer
- Jada can be used either per se, or as a means for designing and implementing higher level coordination languages for the WWW
A Ping-Pong in Jada

_  //--PING--
 TupleClient ts = new TupleClient(ts_host);
 while (true) {ts.out(new Tuple("ping"));
     Tuple tuple = ts.in(new Tuple("pong"));
 }
 Ping ping = new Ping();
 ping.run();

//--PONG--
 TupleClient ts = new TupleClient(ts_host);
 while (true) {ts.out(new Tuple("pong"));
     Tuple tuple = ts.in(new Tuple("ping"));
 }
 Pong pong = new Pong();
 pong.run();
The SHADE Model

- SHADE can be seen as a higher level abstraction of Jada
- Whereas Jada performs singleton level transactions, SHADE is based on multiset rewriting
- Each SHADE object has a name, class and state; the name is the pattern for delivering messages; the type defines the object’s behaviour; the state is the contents of the object’s multiset
A Ping-Pong in SHADE

class ping_class =
{
    in do_ping;
    send pong, do_pong
#    in done;
    terminate
}

class pong_class =
{
    in do_pong;
    send ping, do_ping
#    in done;
    terminate
}

Each class has two methods; when the proper message appears in an object’s multiset, say ping, the method is triggered and sends pong, etc. until a done appears for termination.
GAMMA: Chemical Reactions via Multiset Rewriting

- GAMMA combines the notion of Chemical reactions in CHAM-like models with multiset rewriting

- A program is viewed as a pair (Reaction Condition, Action), and its execution involves replacing those elements in a multiset satisfying the reaction condition by the products of the action

- This process continues until no more such reactions are possible and the system is stable
The GAMMA Rewriting Operator

The above behaviour can be captured by means of the $G$ operator, which is defined as follows:

$$G((R_1,A_1),\ldots,(R_m,A_m))\ (M) =$$

$$\begin{cases}
\text{if } \exists i \in [1,m] \text{ such that } \Box R_i(x_1,\ldots,x_n) \text{ then } M \\
\text{else let } x_1,\ldots,x_n \in M, \text{ let } i \in [1,m] \\
\text{such that } R_i(x_1,\ldots,x_n) \text{ in } \\
G((R_1,A_1),\ldots,(R_m,A_m))\ ((M-\{x_1,\ldots,x_n\})+A_i(x_1,\ldots,x_n))
\end{cases}$$

where $\{\ldots\}$ represents multisets and $(R_i,A_i)$ are pairs of closed functions representing reactions
A GAMMA Example

What the G operator says is that the effect of a pair \((R_i, A_i)\) on a 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 for the elements of \(A_i(x_1, \ldots, x_n)\).

A prime number generator is written as follows:

\[
\text{prime_numbers}(N) = G((R, A))
\]

\[
(\{2, \ldots, N\}) \text{ where}
\]

\[
R(x, y) = \text{multiple}(x, y)
\]

\[
A(x, y) = \{y\}
\]
Other GAMMA Operators

_Transmuter(C, f)_ applies operation f to all the elements of the multiset until no element satisfies the condition C.

_Reducer(C, f)_ reduces the size of the multiset by applying the operation f to pairs of elements satisfying C.

_Optimiser(<, f1, f2, S)_ optimises the multiset according to a criterion expressed through the ordering < between the functions f1 and f2, while preserving the structure S of the multiset.
Other GAMMA Operators  

Expander\((C, f_1, f_2)\), which decomposes the elements of a multiset into a collection of basic values according to the condition \(C\) and by applying \(f_1\) and \(f_2\) to each element \(S(C)\), which removes from the multiset all those elements satisfying \(C\)
Fib in GAMMA

\[
\text{fib}(n) = m \quad \text{where} \\
\{m\} = \sigma(\text{gen}(\{n\})) \\
\text{gen}(N) = G((R_1, A_1), (R_2, A_2))(N) \quad \text{where} \\
R_1(n) = n > 1 \quad A_1(n) \quad R_2(0) = \text{true} \quad A_2(0) = \{1\} \\
\sigma(M) = G((R, A))(M) \quad \text{where} \\
R(x, y) = \text{true} \quad A(x, y) = \{x+y\} \\
\text{fib}(n) = \text{add}(\text{zero}(\text{dec}(\{n\}))) \\
\text{dec} = E(C, f_1, f_2) \\
\quad \text{where} \quad C(x) = x > 1, \quad f_1(x) = x - 1, \quad f_2(x) = x - 2 \\
\text{zero} = T(C, f) \quad \text{where} \quad C(x) = (x = 0), \quad f(x) = 1 \\
\text{add} = R(C, f) \quad \text{where} \quad C(x, y) = \text{true}, \quad f(x, y) = x + y
\]
LO: Linear Logic Meets Multiset Rewriting

Linear Objects view the shared dataspace as a multiset; messages are broadcasted into it and also retrieved by means of a set of interaction rules.

Messages posted to the shared medium are treated as resources which are “consumed” when taken out from it, thus behaving as Linear Logic ops.

<multiset> <broadcast> <built-ins> <goal>
multiset = a1 @ … @ an
broadcast = ^a | ^a @ broadcast
goal= a1 @ … @ an | goal1 & … & goaln | #t | #b
Coordination in LO

The code fragment below is part of a program for the Mastermind game

coder(S) @ current(I) @ ^go(I)  coder(S)
/* coder calls the player ("go(I)") */

coder(S) @ try(I,G) players(N) @ ^result(I,G,A) @
{ answer(S,G,A) next_player(N,I,I1) }
    coder(S) @ current(I1) @ players(N).
/* player I gets answer A to the guess G */

coder(S) @ try(I,G) @ ^victory(I,G) @
{ answer(S,G,G) }  #t.
/* player I has guessed the answer A */
COOLL: Modular LO

COOLL extends LO with modularity and group communication; a program is a set of theories:
\[
\text{theory} \ theory\_name \hspace{1em} \text{method1} \ # \ ... \ # \ \text{methodN}
\]

Communication is either group or broadcast
\[
\text{Communications} = \ ^A | \ ! (\text{dest, msg})
\]
\(\text{dest}\) being the name of a theory to receive \(\text{msg}\)

Methods have the form \(\text{Conditions} \Rightarrow \text{Communications} \Rightarrow \text{Body}\) where the first invokes methods, the second broadcasts and the third changes configurations
The Mastermind in COOLL

```cooll
theory coder

  current(I) => !(players(N),go(I)) => #b

  try(I,G) @{ code(S) @ players(N) } @
    { { answer(S,G,A),
          next_player(N,I,I1) } } => !(players(N),result(I,G,A))
    => #b

  try(I,G) @{ code(S) } @
    { { answer(S,G,G) } } => ^victory(I,G)
    => #t.
```
Synchronizers: Law Enforcers on Objects

- Synchronizers can be seen as the equivalent of the controllers in Law-Governed Linda for the case of an Actor system
- They express coordination patterns by specifying and enforcing constraints that restrict access to a set of objects
- Constraints are defined in terms of object interfaces rather than internal computations performed by them
Xpect and CLF: Coordination for Workflow

- The Coordination Language Facility and its system Xpect is another evolution of LO
- In CLF coordinators coordinate resource manipulations on participants by means of scripting rules
- The LHS of a scripting rule contains tokens which are intended to be removed from participants while the RHS contains tokens to be inserted into the involved participants
CLF Rules and Signatures

_ The rule: \( p(X, Y) \land q(Y, Z) \leftrightarrow r(X, Z) \)

- (i) finds a resource satisfying the property \( p(X, Y) \) and another one satisfying the property \( q(Y, Z) \) for consistent values \( X, Y, Z \)
- (ii) extracts these two resources atomically
- (iii) inserts a resource satisfying \( r(X, Z) \)

_ Signatures are used to denote i-o relationships:

\( p(X, Y) : \rightarrow X, Y \) \quad \( q(X, Y) : X \rightarrow Y \)

Here, \( p \)'s arguments are both output while \( q \)'s first one is input (required) and the second one output
Hotel Reservation in CLF

\_ customer(a,b): \rightarrow a,b \text{ is LOOKUP Agency.customer} \\
\_ roomRes(a,b): a,b \rightarrow \text{ is LOOKUP Agency.roomRes} \\
\_ vacancy(a,b,c): a,b \rightarrow c \text{ is LOOKUP Hotel.vacant}

customer(name,date) @ vacancy(date,"single",no) \\
\text{<> roomRes(name,no)}

\_ Tokens are assigned services, customer generates requests like ("George", "1/1/00"), ("John", "2/2/99"), an instance of the rule is created for every request proceeding in parallel

\_ Further rules are needed to resolve conflicts
Synchronizers

- Based on the Actor model, they are a set of tools able to express coordination patterns within a multi-object language framework
- This is expressed by specifying and enforcing constraints that restrict invocation of objects
- Constraints are defined in terms of object interfaces, rather than internal representations
- An abstract syntax is used, independent of particular languages
A Synchronizer for Dining Philosophers

PickUpConstraint(c1,c2,phil)
{
    atomic((c1.pick(sender) where sender=phil),
            (c2.pick(sender) where sender=phil)),
    (c1.pick where sender=phil) stops
}

The synchronizer enforces atomic access to the two chopsticks \(c_1\) and \(c_2\); when \(phil\) has successfully acquired both chopsticks, the constraint is terminated. The synchronizer applies only to \(pick\) messages (sent by an \(eat\) process)
MESSENGERS: Coordination of Mobile Code for Distributed Systems

A messenger is a message carrying not only data but also a process to manipulate the data.

Each node in a distributed system visited by a messenger executes the process until some navigational command tells it to move elsewhere.

A distributed applications is viewed as a collection of functions whose coordination is managed by a group of messengers.

Inter- and Intra-object coordination is supported.
A Manager-Worker Mobile farm in MESSENGERS

manager_worker()
{
    create(ALL);
    hop(ll = $last);
    while ((task = next_task()) != NULL)
    {
        hop(ll = $last);
        res = compute(task);
        hop(ll = $last);
        deposit(res);
    }
}
A Manager-Worker Mobile farm in MESSENGERS _ 2

_ The Messenger script is injected into the init node of some daemon

_ Logical nodes are created connected to the current node on every neighboring daemon, replicas of the script are created on each node and activated

_ Each Messenger hops back to the original node via the most recently traversed logical link ($last), gets a new task to perform, hops back to its node, does the task and deposits the result back
The *hop* Primitive

\_ \ hop(\ln=n; \ ll=l; \ ldir=d) \\
\_ \ ln \ is \ a \ logical \ node, \ ll \ a \ logical \ link, \ ldir \ the \ link’s \ direction \\
\_ \ (n, l, d) \ is \ a \ destination \ specification, \ n \ being \ an \ address, \ variable \ or \ constant \ (including \ the \ special \ node \ init), \ l \ a \ virtual \ link, \ variable \ or \ constant \ (denoting \ a \ jump \ to \ the \ designated \ node), \ d \ a \ symbol \ denoting \ direction \ ("forward", \ "backward", \ "either")
Compositional Programming

- Shares the same goals with coordination, namely reusability of code, separation of communication from computational concerns, etc
- In a compositional system, the properties of program components are preserved when combined with other components
- Recurring patterns of parallel computation (mergers, streamers) can be identified, isolated and reused
Compositional Programming Derived from Concurrent Logic Programming

- Essentially the CLP formalism is used to express the coordination patterns, the computational parts written in other languages

- Over the years some particular coordination patterns in CLP have been identified and are offered to the user as logical “skeletons”

- These skeletons can be realized in either a concrete CLL (e.g. Strand) or by means of a generic notation (e.g. PCN)
Strand

- Strand is the simplest of CLLs, derived from Parlog and Flat Concurrent Prolog
- It features one-way unification, flat guards, dependent AND-parallelism, list composition and shared single-assignment variables
- The WAM-based implementation of Strand is considered to be one of the fastest in the family of CLLs (the Janus group being the fastest)
- Computational models are written in C, Fortran
Genetic Sequence Alignment Algorithm in Strand (part of the code)

\[
\text{cps}([\text{Seq}|\text{Sequences}], \text{CpList}) :- \\
\text{CpList} := [\text{CPs}|\text{CpList1}], \\
\text{c_critical_points}(\text{Seq}, \text{CPs}), \\
\text{cps}(\text{Sequences}, \text{CpList1}). \\
\text{cps}([], \text{CpList}) :- \text{CpList} := [].
\]

\[
\text{divide}(\text{Seqs}, \text{Pin}, \text{Alignment}) :- \\
\text{Pin} \neq [] \mid \text{split}(\text{Seqs}, \text{Pin}, \text{Left}, \text{Right}, \text{Rest}), \\
\text{align_chunk}(\text{Left}, \text{LAlign}) @ \text{random}, \\
\text{align_chunk}(\text{Right}, \text{RAlign}) @ \text{random}, \\
\text{align_chunk}(\text{Rest}, \text{RestAlign}) @ \text{random}, \\
\text{combine}(\text{LAlign, RAlign, RestAlign, Alignment}). \\
\text{divide}(\text{Seqs}, [], \text{Alignment}) :- \\
\text{c_basic_align}(\text{Seqs, Alignment}).
\]
Program Composition Notation (PCN)

- Strand is a concrete CL language and therefore requires a WAM-based implementation
- PCN is a set of notations, able to express concurrent logic coordination patterns
- Being essentially a set of add-on primitives, PCN can be implemented as an extension of some host computational language (typically C, C++, Fortran, etc.)
- Of course, the system is now also faster
Genetic Sequence Alignment Algorithm in PCN (same code)

```plaintext
  _  cps(sequences,cplist)
    { ? sequences ?= [seq|sequences1] ->
        { ||cplist = [cps|cplist1],
            c_critical_points(seq,cps),
            cps(sequences1,cplist1)
        },
        sequences ?= [] -> cplist = []
    }

divide(seqs,pin,alignment)
  { ? pin != [] ->
      { || split(seqs,pin,left,right,rest),
          align_chunk(left,lalign) @ node(random),
          align_chunk(right,ralign) @ node(random),
          align_chunk(rest,restalign) @ node(random),
          combine(lalign,ralign,restalign,alignment)
      },
      pin == [] -> c_basic_align(seqs,alignment)
  }
```
Compositional Programming Derived from Functional Programming

Here skeletons are realized as higher order functions which represent reusable coordination patterns, related to data partitioning, placement and communication.

The virtues of functional programming (such as compositionality of functions, lack of side-effects, ability to transform programs while preserving their properties, etc.) allow reasoning of programs to be done at the functional specification level.
Configuration Functional Skeletons

distribution (f,p) (g,q) A B
  = align (p _ partition f A)
    (q _ partition g B)

functions f and g specify the partitioning strategy of A and B, respectively, and p and q specify any initial data rearrangement that may be required

partition divides a sequential array into a parallel array composed of sequential subarrays

align pairs subarrays in two distributed arrays together into a new configuration, an array of tupls
Computational Function Skeletons

_ A skeleton for a matrix addition performed in parallel using the previous configuration skeleton:

``` _ matrixAdd A B = (gather _ map _ SEQ_ADD) 
  (distribution fl dl) 
where C = SeqArray ((1..SIZE(A,1)), (1:SIZE(A,2))) 
  fl = [((row_block p),id), 
        ((row_block p),id), 
        ((row_block p),id)] 
  dl = [A,B,C] 
_ Note that SEQ_ADD is defined in some other computational language
```
CoLa: Coordination for DAI

- CoLa is a set of primitives, independent from the host language, especially suited for Distributed AI
- In CoLa one can express communication abstractions (correspondents) and topologies, and local views of computation for a process
- For each process, there is a Range of View which defines the set of correspondents the process can communicate with, and a Point of View indicating the specific communication topology adopted
A Point of View in CoLa

\[
\text{with csTopoVision} \quad \text{-- CoLa base topology class} \\
\text{class csTreeVision is} \quad \text{-- Define Point of View} \\
\text{\hspace{1cm} father(csCor, const csCor);} \quad \text{-- father node in PV} \\
\text{\hspace{1cm} son(csCor, const csCor);} \quad \text{-- son node in PV} \\
\text{end class;}
\]

\[
\text{implementation csTreeVision is} \quad \text{-- Implem of the PVs} \\
\text{\hspace{1cm} son is rule} \quad \text{son(X,Y)} \\
\text{\hspace{1cm} \hspace{1cm} :- csTopoVision.isLinked(X,Y).} \\
\text{\hspace{1cm} father is rule} \quad \text{father(X,Y) :- son(Y,X).} \\
\text{\hspace{1cm} \hspace{1cm} -- Prolog like clauses} \\
\text{end implementation;}
\]
A Range of Vision in CoLa

procedure p(T: in csTreeVision) is
    F: csSet := \{X in T | father(X,self)\};
    -- Compute correspondence
    S: csSet := \{X in T | son(X,self)\};
    myMsgDep :=
        csMsgSendAssDep(highest_prio(S),T,csREAD);
    myMsgId :=
        csMsgAss(myMsgBody,myMsgDep,csFIFO);
    csMsgSend(myMsgId-- Send in the tree topology
    ... ... ... 
end procedure
Combining Task and Data Parallelism

The issue of combining task parallelism with data parallelism is closely related to that of coordinating multidisciplinary applications.

In a way, data parallelism can be viewed as the computational part of a coordination based framework, whereas task parallelism plays the role of communication.

Many of the models proposed here (Braid, Fx, Opus, Orca) use a shared communication medium.
Braid

1. Braid is a data parallel extension to the object-oriented, C++ like, task parallel language Mentat
2. Braid extends Mentat by introducing data parallel Mentant classes, whose objects are partitioned among a number of available processes; operations on these objects are executed in a data parallel way
3. Communication between tasks is achieved by means of shared objects
Combining Task and Data Parallelism in Braid

```cpp
data_parallel mentat class data_par_obj {
    // private member variables
    public:
        int AGG row_sums ROW();
    ...
}

float x, z; int y;
control_par_obj A, B;
data_par_obj my_image;

x = A.op1(); y = my_image.row_sums();
z = B.op1(x, y);
```
Fx

- Fx adds task parallel directives to HPF
- A task corresponds to an execution of a task subroutine, which is a data parallel subroutine where only its actual arguments can be modified
- A number of task subroutines can be invoked within a parallel session (task parallelism)
- Communication is done via a shared medium at procedure boundaries and is the responsibility of the compiler rather than the programmer
Data Parallelism in Fx

```plaintext
template t(n)
align A(i,j) with t(i)
align B(i,j) with t(j)
distribute t(cyclic)
do i=1,n
   A(i,:) = A(i,:) + B(:,i)
endo
```

The `template`, `align`, and `distribute` directives are used to distribute the rows of A and the columns of B cyclically across the node array; loop iterations are independent and can execute in parallel.
Task Parallelism in Fx

begin parallel
  do i=1,10
    call src(A,B) output:A,B
    call p1(A) input:A output:A
    call p2(B) input:B output:B
    call sink(A,B) input: A,B
  enddo
end parallel

Forty tasks are created, in each iteration src sends the two arrays to p1 and p2, the latter after operating on them pass them to sink; p1 and p2 can execute in parallel, and pipelining is supported
Opus

- A coordination superlanguage for HPF, where processes communicate via a *Shared Abstraction (SDA)*, a Linda-like common forum
- An SDA is in fact an ADT, containing data specifying its state and methods for manipulating this state
- SDAs can be used either as data servers between concurrently executing processes (data par) or as computation servers driven by a controlling task
A Data Server for a FIFO Bounded Buffer in Opus

```fortran
_ SDA TYPE buffer_type(size)
   INTEGER :: size
   REAL, PRIVATE :: fifo(0:size-1) ! FIFO buffer
   INTEGER, READ ONLY :: count=0 ! no of full elements
   INTEGER, PRIVATE :: px=0 ! producer index
   INTEGER, PRIVATE :: cx=0 ! consumer index
...
CONTAINS
   SUBROUTINE put(x) WHEN (count .LT. size)
     ! implementation in Fortran
   END
   SUBROUTINE get(x) WHEN (count .GT. 0)
     ! implementation in Fortran
   END
...
END buffer_type
```
Managing the Task Parallelism for the FIFO Buffer

PROCESSORS R(128)

SDA(buffer_type)::buffer1, buffer2

... CALL buffer1%CREATE(256) ON PROCESSORS R(1)
CALL buffer2%CREATE(1024) ON PROCESSORS R

Each one of the two CREATE statements generates an SDA with some buffer size, the first goes to R(1) and the second to the rest of the processors; buffer1 and buffer2 are used as handles.
Orca

- Orca is quite similar to Opus and can be seen as an object-based DSM system
- Communication in Orca is done via shared objects, instances of ADTs, by applying user-defined ADT operations on them
- Data parallelism is expressed through partitioned objects, containing arrays that can be distributed among multiple processors
Mixed Task and Data Parallelism in Orca

PROCESS Worker(P,M1,M2:integer; procs:CPUlist)
  A: fftObject[1..N];
BEGIN
  A$partition(N);
  A$distribute_on_list(Procs,P,BLOCK);
  FOR i IN M1..M2 DO readmatrix(i,A); 2Dfft(A); OD;
END;

PROCESS Coordinator(Ncpus,NWorkers,NMatrs: integer);
  P,S: integer;
BEGIN
  P:=Ncpus/NWorkers; S:=NMatrs/NWorkers;
  FOR i IN 0..NWorkers-1 DO
    FORK Worker(P, i*S, (i+1)*S-1, [i*P..(i+1)*P-1])
     ON (i*P); OD;
END;
Control-Driven Coordination Formalisms

Diagram:

- **Pi**: Input
- **Po1**: Output 1
- **P02**: Output 2
- **Prod**: Product
- **e1, e2**: Events
- **C1i**, **C1o**: Channels 1 Input, Output
- **Cons1**: Consumer 1
- **C2i1**, **C2i2**, **C2o**: Channels 2 Input 1, Input 2, Output
- **Cons2**: Consumer 2
Main Characteristics of Control-Driven Coordination Formalisms

_ Processes communicate in a point-to-point manner by means of well defined interfaces, usually referred to as (input or output) _ports_ which are used to set up _streams_ or _channels_

_ Changes to such network configurations are often triggered by means of raising and observing _events_

_ Processes are normally treated as black boxes and the actual data being exchanged do not affect the state of the computation_
Configuration and (Dynamic) Reconfiguration Languages

- Configuration Management is effectively control-driven coordination for the following reasons:
  - Configuration languages are separate entities from computational ones used to implement components
  - Components are context independent and specify visible behaviour via well defined interfaces
  - Complex components are definable as composition of simpler ones
  - Changes are expressed at the configuration level as changes to component instances or interconnections
Architecture Description Languages (ADL) and Software Architectures

ADLs and Software Architecture languages is yet another way of viewing coordination, because they share a number of common requirements such as: component abstraction and composition, communication abstraction, ability to model dynamic behaviour, etc.

Often, components are represented as black boxes, with ports as interfaces being connected by means of connectors (the control-driven paradigm)
Proteus Configuration Language

- In PCL the unit of configuration is a family entity, representing one or more versions of a component.
- A family entity comprises a composition structure, a type, attributes, a parts section specifying its composition, and version descriptors.
- A component may have a number of ports, signifying offered and required services.
- Evolution is specified by version descriptions triggered by action ports.
Specifying Components in PCL

family nurse inherits application_component is
  classification
    REALISATION => concrete
  end
attributes
  persistent_state = true
  monitors: integer range 0..3
end
parts
  IN_PORTS => alarm_in[monitors]
  OUT_PORTS => data_out[monitors]
    quit -- action port for removal
end
end
Dynamic Reconfiguration in PCL

_  version initial_pms of ward is
    attributes
      nurses := 3, monitors := 4
    relationships
      CB1:component_binding => nurse[1], monitor[1]
      CB2:component_binding => nurse[2], monitor[2]
      CB3:component_binding => nurse[3], monitor[3, 4]
  end

version nurse[3]_quit inherits initial_pms of ward is
    attributes
      nurses := 2
    relationships
      CB2:component_binding => nurse[2], monitor[3, 4]
  end
Conic

Conic is based on a variant of Pascal and features *logical nodes* configured together by means of *links* established between *entry* and *exit ports*.

Logical nodes are system configuration units, comprising sets of tasks executing concurrently; sets of nodes form *groups*.

Dynamic reconfiguration is limited; evolutions must be completely specified at compile time and the system must be quiescent or info may be lost.
Specifying Components in Conic

```conic
/_ group module patient;
  use monmsg: bedtype, alarmstype;
  exitport alarm: alarmstype;
  entryport bed: signaltype reply bedtype;
  << code >>
end.

group module nurse;
  use monmsg: bedtype, alarmstype;
  entryport alarm[1..maxbed]: alarmstype;
  exitport bed[1..maxbed]: signaltype
    reply bedtype;
  << code >>
end.
```
Dynamic Reconfiguration in Conic

```plaintext
    system ward;
    create
        bed1: patient at machine1;
        nurse: nurse at machine2;
    link
        bed1:alarm to nurse.alarm[1];
        nurse.bed[1] to bed[1].bed;
    end.
manage ward;
create
    bed2: patient at machine1;
unlink
    bed1:alarm from nurse.alarm[1];
    nurse.bed[1] from bed[1].bed;
link
    bed2:alarm to nurse.alarm[1];
    nurse.bed[1] to bed[2].bed;
end.
```
Darwin & Regis

Darwin and its associated system Regis extend Conic in a number of ways:

- Language independence
- Stronger notion on dynamic reconfiguration by means of direct component instantiation specified at run-time
- Components can interact through user-defined communication primitives
- Input ports of a component are *provided* to the environment for other processes to post there data and output ones are *requiring* a port reference to post data
Specifying Components in Darwin

```c
 component supervisor (int w)
 { provide result <port,double>;
   require labour <component,int,int,int>;
 }
 component worker (int id, int nw, int intervals)
 { require result <port,double>;
 }
 component calcpi2(int nw)
 {
   inst
     S:supervisor(nw);
   bind
     worker.result -- S.result;
     S.labour -- dyn worker;
 }
```
Dynamic Reconfiguration in Darwin

```cpp
_ supervisor::supervisor(int nw)
{
    const int intervals=400000;
    double area=0.0;
    for (int i=0; i<nw; i++)
    {
        labour.at(i);
        labour.inst(i,nw,intervals);
    }
    for (int i=0; i<nw; i++)
    {
        double tmp;
        result.in(tmp);
        area+=tmp;
    }
    printf("Approx pi %20.15lf\n",area);
}
```
Components consist of *application tasks* featuring *input-output ports* and communication *channels*. At run-time, tasks create *processes* and channels create *links*. Dynamic reconfiguration is done by raising and observing *events*; however, as in Conic, unrestricted dynamic creation of tasks is not possible; also, before breaking a link, a task must raise a *safe* message so that data is not lost.
Specifying Components in Durra

```
_ task producer
  ports
    output: out message;
  attributes
    processor="sun4";
    procedure_name="producer";
    library="/usr/durra/srclib";
  end producer;

task consumer
  ports
    input: in message;
  attributes
    processor="sun4";
    procedure_name="consumer";
    library="/usr/durra/srclib";
  end consumer;

channel fifo(msg_type:identifier, buffer_size:integer)
  ports
    input: in msg_type; output: out msg_type;
  attributes
    processor="sun4"; bound=buffer_size;
    package_name="fifo_channel";
    library="/usr/durra/channels";
  end fifo;
```
Dynamic Reconfiguration in Durra

```java
  _
task dynamic_producer_consumer
  components
  p: task producer; c[1..2]: task consumer;
  buffer: channel fifo(message,10);
structures
  L1: begin
    baseline p, c[1], buffer;
    buffer: p.output >> c[1].input;
    end L1;
  L2: begin
    baseline p, c[2], buffer;
    buffer: p.output >> c[2].input;
    end L2;
reconfigurations
  enter => L1;
  L1 => L2 when signal(c[1],1);
end dynamic_producer_consumer;
```
The Programmer’s Playground

This configuration formalism is based on the notion of I/O abstractions, where each module has a presentation that consists of data structures that may be externally observed and/or manipulated.

An application consists of a set of modules and a configuration of logical connections among the data structures in the module presentations.

Updating data structures, causes communication to occur implicitly based on the logical connections.
Dynamic Reconfiguration in the Programmer’s Playground

This is achieved by means of logical handles, which define virtual connections between input and output “ports”.

Every logical handle defines an i-o relationship between a number of ports; associating a handle with another one, effectively creates a set of virtual links between the two groups of ports.

At a physical level, however, connections are implemented as being point-to-point.
The Graphical Form of a Program in the Programmer’s Playground
Textual Programming in the Programmer’s Playground: Producer

```cpp
#include "PG.hh"

PGint next=0;
PGstring mess;

send_next(PGstring mess, static int i)
{ if (strcmp(mess,"ok")==0) next=i++; }

main()
{
    PGinitialise("producer");
    PGpublish(next,"next_int",READ_WORLD);
    PGpublish(mess,"ok",WRITE_WORLD);
    while (1)
        PGreact(mess,send_next);
    PGterminate();
}
```
Textual Programming in the Programmer’s Playground: Consumer

```cpp
#include "PG.hh"

PGint next=0;
PGstring mess;

void consume_int(PGint i)
{ /* consumes list of integers */ }

main()
{
    PGinitialise("consumer");
    PGpublish(mess,"ok",READ_WORLD);
    PGpublish(next,"next_int",WRITE_WORLD);

    while (1)
    { PGreact(next,consume_int); mess="ok"; }
    PGterminate();
}
```
Olan

_ Olan is an object-oriented configuration language where a configuration is viewed as a hierarchy of interconnected *components*

_ Components have well-defined interfaces called *services*, either *provided* or *required*; service exchange is realized by means of *connectors*

_ *Notifications* are used as an event broadcasting mechanism (through connectors though), which can cause *reactions* by the observing processes
Graphical Presentation of Olan Components
Textual Presentation of Olan Components

```
component class UserSession
  interface
    require SendOp (in operation);
    provide ReceiveOp (in operation);
    ...
  implementation
    theCont = dyn inst CoopController; // dynamic inst
    theAppl = inst SharedAppl;

    // mapping using methodcall connector
    connector
      ReceiveOp => theCont.ReceiveOp;
      theCont.SendOp => SendOp;

    // interactions
    theCont.Launch => theAppl.Init;
    theCont.SendOp => theAppl.ReceiveOp;
```
C2

_ C2 is a style for building systems with complex user interfaces

_ Architectures consist of components (written in any language) and connectors
  – Architecture is layered
  – Connectors broadcast messages up or down one layer
  – Request messages only go up; Notifications only go down
  – Components connect to one connector above and one below
A Graphical C2 Example

Database Component

Admin IU  User IU

Window System
A Textual C2 Component Description

_ component StackADT is
  interface
    top_domain in null; out null;
    bottom_domain
      in  PushElement (value : stack_type);
      out ElementPushed (value : stack_type);
  parameters null;
  methods
    procedure Push (value : stack_type);
    function Pop () return stack_type;
  behavior
    received_messages PushElement;
    invoke_methods Push;
    always_generate ElementPushed;
end StackADT;
In UniCon, a system is comprised of (possibly nested) components which have, among other things, a number of players (i.e. ports).

In addition, there exist connectors associated with roles, that specific named entities of the components (i.e. the players) must play.

Both components and connectors are defined in terms of a name, a type, an interface and an implementation.
A Primitive Component and Connector in UniCon

COMPONENT Sort

INTERFACE IS

TYPE Filter
PLAYER input is StrIn
SIGNATURE(“line”)
PORTBINDING(stdin)
PLAYER output is StrOut
SIGNATURE(“line”)
PORTBINDING(stdout)
IMPLEMENTATION IS
VARIANT sort IN “sort”
IMPLTYPE (Executable)
INITACTUALS (“-f”)

CONNECTOR Unix-pipe

INTERFACE IS

TYPE Pipe
ROLE source is source
MAXCONNS(1)
ROLE sink is sink
MAXCONNS(1)
IMPLEMENTATION IS
BUILTIN
END
Architecture Description Interchange Language (ACME)

- ACME features the usual constituents of an ADL: components forming systems, and communicating by means of connectors via (typed) ports
- There are roles associated with connectors, e.g. an event broadcast connector has an event-announcer role and a number of event-receiver roles
- There may be more than one description for some component; rep-maps are then used to associate internal representations with external interfaces
An ACME Component System
Textual ACME

System simple_cs = {
  Component client = { Port send-request; };
  Component server = { Port receive-request; };
  Connector rpc = { Roles { caller, callee} }; 
  Attachments {
    client.send-request to rpc.caller;
    server.receive-request to rpc.callee;
  }
}
Rapide

- Rapide, features components defined by means of interfaces, connections and constraints
- Interfaces define the behaviour of components, connections define communication among components using only those features specified by their interfaces, and constraints restrict the behaviour of components and interfaces
- There exist events which can be parameterized with types and values
A Producer-Consumer Scenario in Rapide _ 1

_ type Producer(Max: Positive) is interface
  action out Send(N: Integer);
  action in  Reply(N: Integer);
behavior
  Start => Send(0);
  (?X in Integer) Reply(?X)
    where ?X < Max => Send(?X+1);
end Producer;

type Consumer is interface
  action in  Receive(N: Integer);
  action out Ack(N: Integer);
behavior
  (?X in Integer) Receive(?X) => Ack(?X);
end Consumer;
A Producer-Consumer Scenario in Rapide

architecture ProdCons() return SomeType is
    Prod: Producer(100);
    Cons: Consumer;

connect
    (?n in Integer)
    Prod.Send(?n) => Cons.Receive(?n);
    Cons.Ack(?n) => Prod.Reply(?n);
end architecture ProdCons;

The coordinator above creates two process instances, one for producer and one for consumer, and associates the output event of the former with the input event of the latter and vice versa.
POLYLITH

- POLYLITH is effectively a MIL, enhanced with functionality usually found in coordination languages such as ports and events
- Components are modules with interfaces for each communication channel upon which running instances of a module will exchange messages
- An abstract decoupling agent, a *software bus*, is used as a communication channel; processes can be “plugged” into or “unplugged” from it
A Producer-Consumer Example in POLYLITH _ 1

```c
main(argc,argv) /* a.c (exec in a.out) */
{
    char str[80];
    ...
    mh_write("out",...,"msg1");
    ...
    mh_read("in",...,str);
    ...
} 

service "A":
{
    implementation:
    {
        binary: "a.out"
    }
    source "out": {string}
    source "in": {string}
}

main(argc,argv) /* b.c (exec in b.out) */
{
    char str[80];
    ...
    mh_read("in",...,str);
    ...
    mh_write("out",...,"msg2");
    ...
} 

service "B":
{
    implementation:
    {
        binary: "b.out"
    }
    source "out": {string}
    source "in": {string}
}
```

IPA'99 120
A Producer-Consumer Example in POLYLITH _ 2

_ orchestrate “example”:
  {
    tool “foo”: “A”
    tool “bar”: “B”
    tool “bartoo”: “B”
    bind “foo out” “bar in”
    bind “bar out” “bartoo in”
    bind “bartoo out” “foo in”
  }

_ Using the primitives mh_read and mh_write, the two modules send a message msg1 or msg2 to their out (source) ports and receive a message of type string into their in (sink) ports
TOOLBUS

TOOLBUS is similar to POLYLITH, in that a software bus *(toolbus)* is used as the communication medium. The building block is a "tool"; types and numbers of tools must be defined statically; however, any number of instances of these tools may be active. Adapters are used for data compatibility between different tools; configurations are expressed in terms of *T-scripts* which can be formally reasoned.
Coordinating a Set of Tools in TOOLBUS

Define COMPILER =
( rec-msg(compile,Name) . snd-eval(compiler,Name) .
  ( rec-value(compiler,error(Err),loc(Loc)) .
    snd-note(compile-error,Name,error(Err),loc(Loc))
  ) * rec-value(compiler,Name,Res) .
  snd-msg(compile,Name,Res)
) * delta
Define EDITOR =
subscribe(compile-error) .
(rec-note(compile-error,Name,error(Err),loc(Loc)) .
snd-do(editor,store-error(Name,error(Err),loc(Loc))) +
rec-event(editor,next-error(Name)) .
snd-do(editor,visit-next(Name)) .
snd-ack-event(editor)
) * delta
COCA: Groupware Using Coordination

COCA is a Prolog-based groupware coordination language, featuring \textit{channels} and \textit{roles}.

Communication is realized using \textit{IP multicast}, the metaphor being a dual bus architecture:

- A \textit{collaboration bus} connects all participants and provides basic communication among them.
- A \textit{conference bus} connects a local instance of the COCA virtual machine with the various collaboration tools (video, audio, drawing, etc.); the VM is also connected to the collaboration bus.
A Presentation Scenario in COCA

_ collaboration “presentation”
{ collaboration-bus { channel(remote). }
role “slide viewer”
{ conference-bus { channel(l-in), channel(l-out) }
on-arrive(gate(remote),id(URL),slide(X)) :-
  l-out(id(URL),slide(X)).
on-arrive(gate(l-in),id(URL),slide(X)) :-
  remote(id(URL),slide(X)).
}

_ COCA VMs communicate via remote, and actors within some VM communicate via the local i/o channels, l_out to display received slides and l_in to forward local ones for remote display
MANIFOLD

_ MANIFOLD is a “traditional” coordination language (i.e. not an ADL, configuration language, etc.) featuring *ports*, *events* and *streams*

_ Coordinators are written in a fully fledged control-driven language which clearly separates the coordination patterns from the computational ones

_ Changes to a system are done dynamically by *state transitions* triggered by observing raised events
A Conferencing System Modelled in MANIFOLD _ 1

_event leave.
manifold Session(event) import.
manifold Connect(process p1, process p2)
{
  begin: (p1 -> p2, p2 -> p1, terminated(self)).
  leave.p1|leave.p2: .
}
manifold Participate(process me)
{
  ignore join.me.

  begin: while true do
  {
    begin: terminated(self).
    join.*other: Connect(me,other).
  }.
  leave.me: .
A Conferencing System Modelled in MANIFOLD_2

manifold User()
{
    event remove_me.
    begin: (Participate(self),
        raise(join),
        Session(remove_me),
        terminated(self)).
    remove_me: raise(leave).
}

Session implements the actual conferencing activity, Connect establishes a communication link between two processes, Participate creates a Connect process for every new User
ConCoord

- ConCoord can be seen as a “structured” MANIFOLD, comprising the same structures as the latter; events can be parameterized with data
- Coordinators in ConCoord are created in a hierarchical manner, where inner ones have no access to outer ones, in terms of observing raised signals or establishing i/o connections
- State changes are communicated by message passing and communication can be synchronous
A Pipeline of Generic Processes in ConCoord

coordinator <t_node, t_data> gen_dyn_pipeline
{
inport <t_data> in;
outport <t_data> out;
states error(), done();
create t_node n bind in -- n.left, n.out -- out;
loop
{ choose
{
  sel(t_node n | n.new and not n.right--)
  => create t_node new_n
     bind n.right -- new_n.left, new_n.out -- out;
  sel(t_node n | n.new and n.right--)
  => error();
}
}
Little-JIL

- A graphical coordination language featuring:
  - Explicit resource specification and dataflow
  - Proactive and reactive control constructs
  - Hierarchical decomposition of steps which are the central abstraction in the language
  - Control flow operators restrict execution of substeps
    » Sequential & Parallel (AND), Choice & Try (OR)
  - Processes are viewed as agents who need coordination
  - Precondition and postcondition guards
  - Exception Handling
Little-JIL Step Notation

- Interface Badge
- Precondition Badge
- Postcondition Badge
- TheStepName
- Control Flow Operator
- Reactions
- Exception Handlers
Agent Coordination in Little-JIL
Cooperative Systems Design Language (CSDL): Cooperation Using Coordination

- CSDL supports the definition of the coordination aspects of a cooperative system

- **Configurations** comprise users and applications managed by *coordinators*; the latter consist of:
  - a *specification* defining groups and cooperation policies in terms of *requests* exported selectively to members of different groups
  - a *body* defining *access rights* associated with groups
  - a *context* defining coordinator dependencies in terms of groups mapping
A Cooperative System in CSDL
Specifying ab X-Windows Coordinator in CSDL

```
coordinator XWindow
{  group ConnectedUsers;
    group Output nestedIn ConnectedUsers;
    group Input nestedIn Output;
    invariant #Input <= 1;
    requests
    { exportedTo extern
        { join Output other
            { actions: insert ConnectedUsers other;
                insert Output other; }
            join Input other
            { requires: other in Output and #Input = 0;
                actions: insert Input other; }
        } leave Output other
        { actions: extract Output other;
            extract ConnectedUsers other; }
    }
}
```
Specifying ab X-Windows Coordinator in CSDL _ 2

coordinator body XWindow
{
  S: switcher inOut XSwitcher;
  group ConnectedUsers { connected; inOff; outOff; }
  group Output { outOn; }
  group Input { inOn; }
}

The specification includes an invariant stating constraints on group cardinality and membership, and requests (join, leave) exported selectively.

Exchange of information is done via logical switches that model multiplexing of data streams.
Conclusions _ 1

_ In the traditional coordination languages, there will be further work and convergence on the issue of the communication medium: structured and localized, but not unrestricted broadcasting or point-to-point

_ With the development of Internet technologies, there will be a new class of coordination formalisms, especially suited for that purpose, such as XML
Conclusions _ 2

_ Also, new application areas will be explored such as Cooperative Information Systems, Open and Distributed Multimedia, E-Commerce, Mobile Computing, etc.
_ Thus, there will be a closer collaboration between traditional coordination programming groups and ones using coordination in a different setting (CSCW, workflow, collaboration, groupware, etc.)
_ Coordination will (successfully?) marry CORBA