ENGINEERING SYSTEMS with COORDINATION MODELS and TECHNOLOGIES

Alessandro Ricci

(DEIS, Università di Bologna in Cesena)

aricci@deis.unibo.it

Goals

- Recalling some elements of distributed system engineering
- Introduction to coordination models and technologies
- Overview of the TuCSoN coordination model and technology
  - Experimenting TuCSoN on-the-fly
- TuCSoN for students’ projects? Why not..

Outline

- (PART I) Elements of distributed system engineering
  <few important points>
- (PART II) Introduction to the coordination world
  <quick overview>
  – Tuple Space / Linda example
- (PART III) TuCSoN model and technology
  <focus>
  – (NOTES 1) TuCSoN Live
  – (NOTES 2) TuCSoN model/language details
  – (NOTES 3) Some coordination patterns in Linda and TuCSoN
- (PART IV) Discussion: using TuCSoN for your projects
  <focus according to available time>

PART I

Elements of distributed systems engineering

Recalling some basic engineering principles

- Engineering approach
  - Using methodologies / models / languages / technologies to face problems of a certain domain at the *proper level of abstraction*
    - direct/synthetic description of (1) domain aspects and their relationships and (2) aspects/relationships evolution
    - Pervasive principle: *Encapsulation*
  - Lack of good abstractions leads to lower level *mechanisms*
    - Longer design/development time
    - Evolution is a nightmare
      - Any addition, change --> new aspect --> new mechanism

Concurrent/Distributed system scenarios

- Space, Time, Interaction
- Concurrency/parallelism
  - Multiple independent activities / loci of control running simultaneously
- Distribution
  - Activities running on different and heterogeneous execution contexts (machines, devices, ...)
- Interaction
  - *Dependencies* among activities
  - Collective goals involving activities coordination/cooperation

Object-Oriented paradigm: not enough

- How to model *an independent activity*?
  - Objects? No way
    - Objects encapsulate a state and a behaviour, but not a control flow
      - Objects have autonomy over their state, they can control it
      - Objects have not autonomy over their behaviour, they cannot control it
      - Control flows along with data, going inside out though objects by means of method invocation (as reification of message passing)
    - Control is outside objects, owned by human designer who acts as a control authority, establishing the control flow
    - Object interaction is limited and disciplined by interfaces, governed by the human designer
- How to model *concurrent* activities? How to model *interaction and coordination* among concurrent activities?
  - Method invocation?? No way!

Toward task-oriented approach

- Identifying system basic *tasks*
- A task can be assigned to a single *agent* (individual task) or to a *society* of agents (collective task)
  - The *agent* abstraction
    - Entity responsible of carrying out an individual task
    - Encapsulates a control flow
      - needed to carry out the task
  - Society of agents and the *coordination artifact* abstraction
    - Societies as set of agents using a *coordination artifact* to carry out the social task
    - Coordination artifacts as the abstractions modeling the shared means/tools used by agents to fulfill a collective task
      - Coordination artifacts in human society: languages, protocols, as well as semaphors, ticket dispensers, blackboards, schedulers, forms, …

Basic engineering approach

- [1] Task identification
  - Individual tasks
    - (Assigned to)/(drive the design of) single agents
  - Social tasks
    - (Assigned to)/(drive the design of) agent societies
      - Coordination artifacts
- [2] Task folding/unfolding
  - Task unfolding
    - From individual to social
  - Task folding
    - From social to individual

Some design

- Agent design issues
  - Information/Permissions required to fulfill the task
  - Information possibly expected by task fulfillment
  - Behaviour to fulfill the task
  - Topology (agent location)
  - ..
- Coordination artifacts design issues
  - Coordination laws required to fulfill the collective task
  - Shared communication protocols
  - Constraints ruling the collective task
  - Global properties result of the collective task
  - Topology
    - Location for *embodied/physical* coordination artifacts
  - ..

An example: an alarm system

- Scenario:
  - a building where access to rooms/resources must be ruled according to some global organisation policy
- Example of an individual task:
  - Monitoring a room
  - Identifying users
  - Informing the police
  - ..
- Example of a collective task
  - Organisation/Security policies
    - When detecting unidentified users in rooms X,Y, lock resources R1, R2 and inform the police
    - From 23.00 to 6.00 only some users are allowed to access the building

Downto development / deployment

- Agents and coordination artifacts need *infrastructures*
  - Providing basic services to support agents and artifacts at runtime
    - Services for agent life cycle (creation, execution, death, migration..)
    - Services for coordination artifacts enaction, management, use, evolution
      - Enabling communication, synchronisation,..
  - *keeping the abstractions alive* motto
    - Basic services to deal with agent and coordination artifacts as first class issues at runtime
      - Support for dynamic construction, observation, evolution of the systems
  - Despite of the technology used for developing actually agents and artifacts

<at the blackboard: Infrastructure picture>

Technology picture (I)

- Web (HTTP+CGI/JSP/ASP/Servlets..)
- Distributed Object Model (Java RMI, CORBA, DCOM+/.NET...)
- Distributed Component Model (CORBAcomponents, EJB, .NET)

> weak support for task-oriented approach

Technology picture (II)

- Service-Oriented Infrastructures (Web Services, CORBAservices, Jini, OSGi,..)
- Agent-oriented (JADE,RETSINA,TuCSoN..)
- Coordination Technologies (TuCSoN, JavaSpaces...)

> good support for task-oriented approach

Focus on coordination models and technologies

- Good support for task-oriented approach
  - Promote separation between individual and collective tasks
  - Provide explicit abstractions to model/develop coordination artifacts
  - Support the agent abstraction

PART I - SUMMING UP

- The engineering of distributed systems calls for abstractions encapsulating the control flow
- Task-oriented approach
  - Individual and collective tasks
  - agents and coordination artifacts abstractions
- Infrastructure concept
  - Supporting development and deployment
  - Several technologies

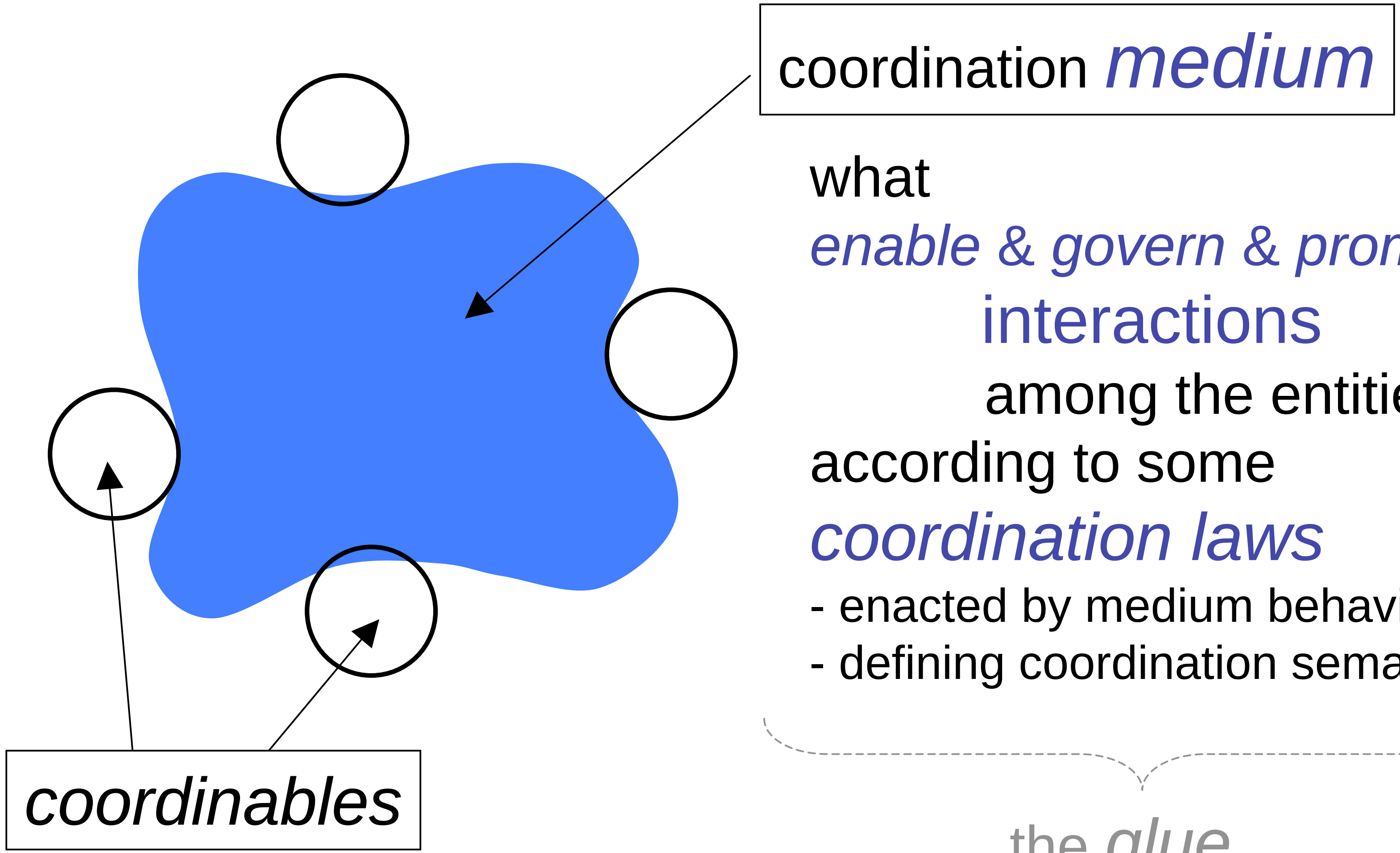
PART II

Coordination models and technologies

Coordination models & languages roots

- Concurrent/parallel programming context (~1980s)
  - Programming = Computation + Coordination
    - Coordination as the glue that binds separate activities into an ensemble (Gelernter)
- Software engineering context (~1990s)
  - Coordination = constraining/promoting/managing interaction among independent components
  - Architectures = Components + Connectors

Coordination metamodel



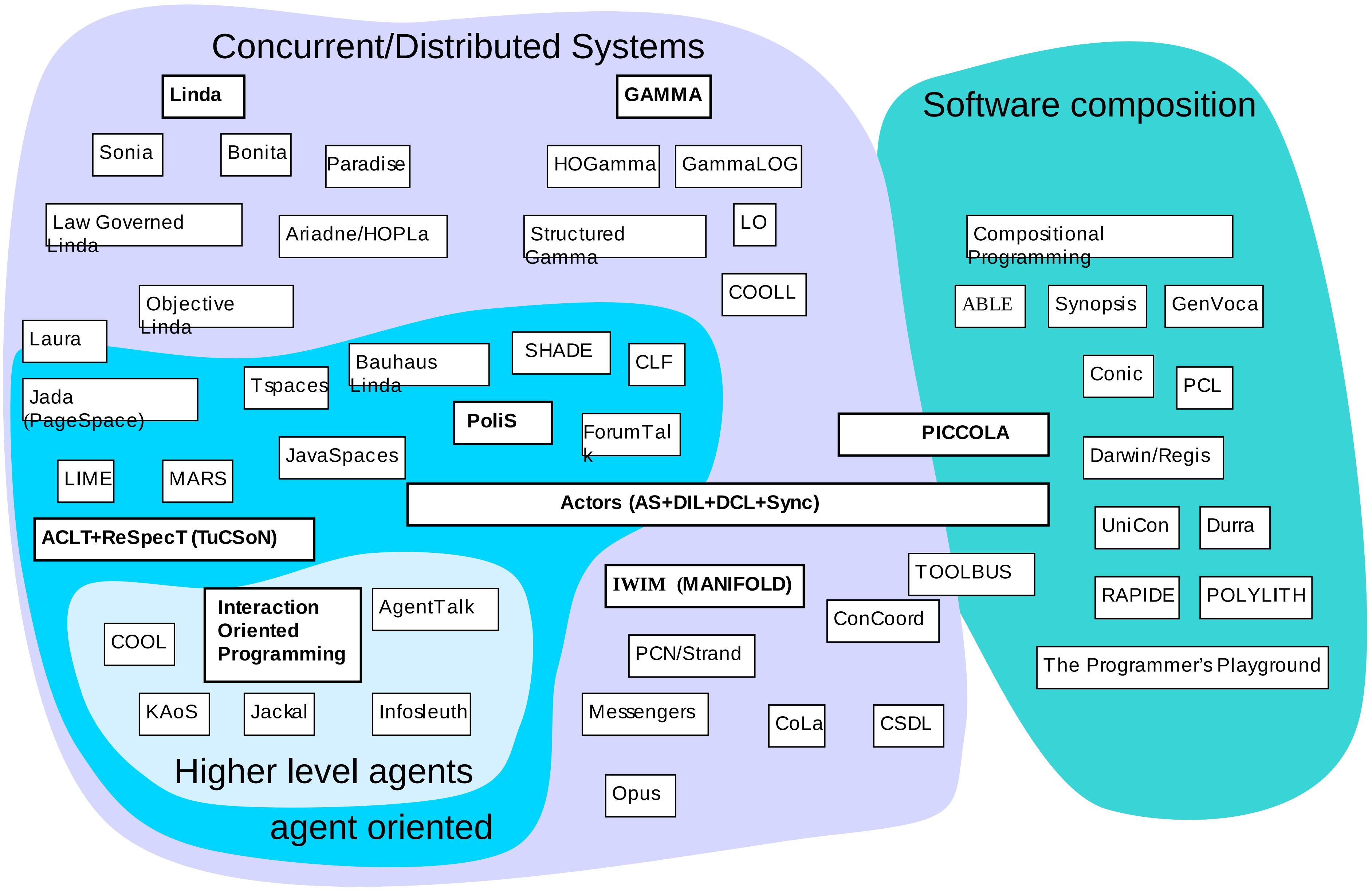
what
enable & govern & promote
interactions
among the entities
according to some
coordination laws
- enacted by medium behaviour
- defining coordination semantics

$\underbrace{\hspace{8cm}}_{\text{the } \textit{glue}}$

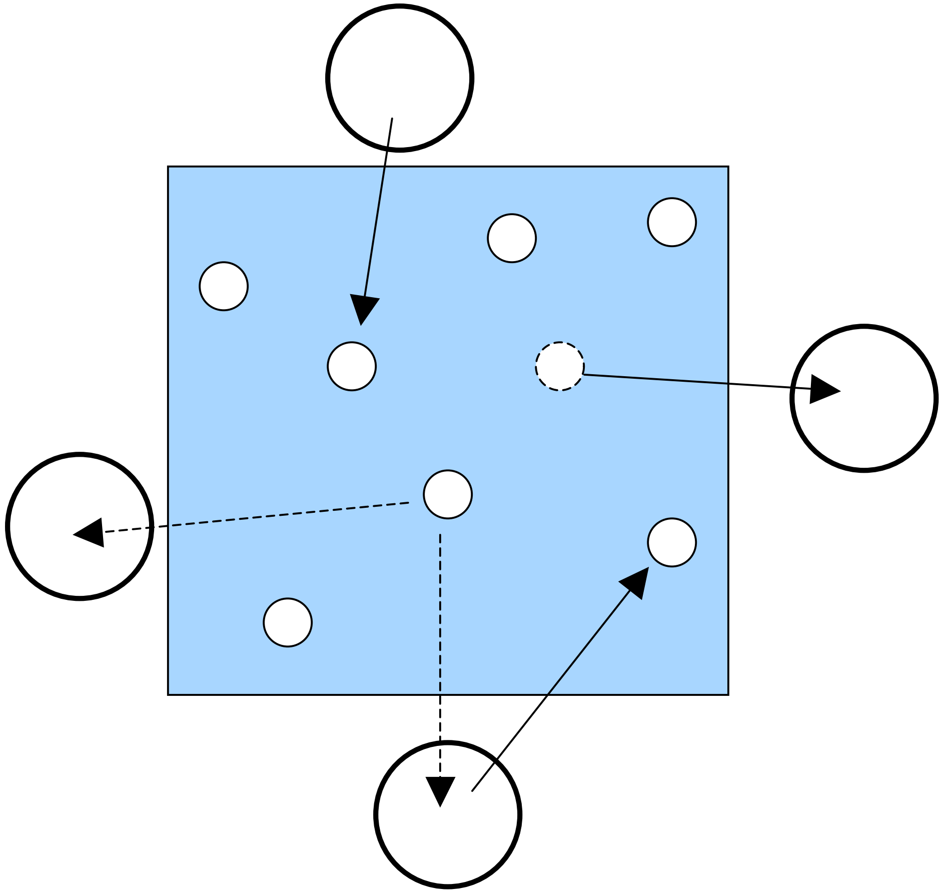
Coordination metamodel (Ciancarini, 1996)

- *Coordinables*
  - Entities whose mutual interaction is ruled by the model
  - Example: processes, threads, objects, users, *agents*…
- *Coordination Media*
  - Abstractions enabling and ruling agent interactions
  - The core around which the components of the system are organised
  - Examples: semaphors, monitors, channels, tuple spaces, blackboards, pipes
- *Coordination Laws*
  - Define the behaviour of the coordination media in response to interaction events
  - Usually defined in terms of
    - The *communication language*
      - the syntax used to express and exchange data structures
      - Examples: Tuples, XML pages, Logic Tuples, (Java) Objects, ….
    - The *coordination language*
      - Set of interaction primitives and their semantics
      - Examples: in/out/rd.. (Linda), send/receive (channels), push/pull (pipes..)

Model/Language Families (1999)



The Tuple Space model



Coordinables synchronise, cooperate, compete based on tuples available in the tuple space, by associatively accessing, consuming and producing tuples

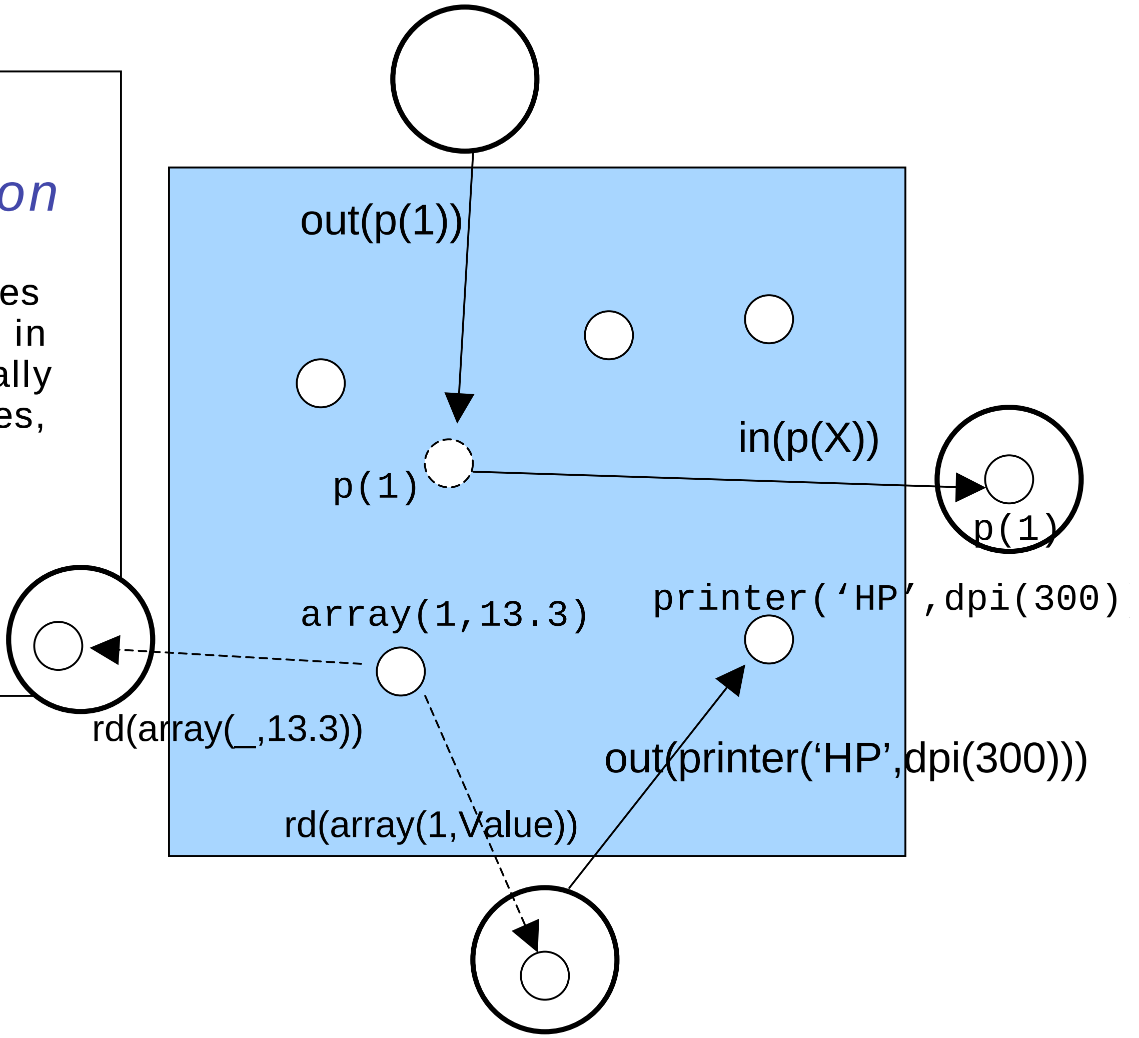
- Coordination medium: Tuple Space
  - Multiset / bag of data object/structures called *tuples*
- Communication Language: tuples
  - Tuple = ordered collection of (possibly heterogeneous) information items
- Coordination Language → set of operations to put and retrieve tuples to/from the space

A language for Tuple Spaces: Linda

- Communication Language
  - Tuple, Templates (anti-tuples) and tuple matching
    - Examples: p(1), printer(‘HP’,dpi(300)), my_array(0,0.5), matrix(m0,3,3,0.5), tree_node(node00,value(13),left(_),right(node01)), …
- Coordination primitives
  - out(T)
    - Puts in the space the tuple T
    - Examples: out(p(1)), out(printer(‘HP’,dpi(300)), out(array(1,13.4)), out(course(‘Denti Enrico’,‘Poetry’,hours(150))…
  - in(TT)
    - Removes from the space a tuple matching the template TT
      - Blocking behaviour
      - non-determinism
    - Examples: in(p(X)), in(printer(Name,dpi(300)), in(array(1,Value))…
  - read(TT)
    - Reads (without removing) from the space a tuple matching the template TT
      - Blocking behaviour
      - Non-determinism
    - Examples: read(p(1)), readd(printer(‘HP’,dpi(Dpi)), read(array(Index,13.4)), read(course(‘Denti Enrico’,Course,_), read(course(_,’Poetry’,Hours))…

Tuple Spaces/Linda features

- *Generative Communication*
  - until explicitly withdrawn, the tuples generated by coordinables have an independent existence in the tuple space. A tuple is equally accessible to all the coordinables, but is bound to none
- Associative access to the tuple space



Generative communication properties

- Communication orthogonality
  - Both senders and the receivers can interact even without having prior knowledge about each others
  - → Space uncoupling (also called distributed naming)
  - → Time uncoupling
- Free Naming
  - → Support for continuation passing, Structured naming and inverse Structured naming
    - Flexibility exploiting tuple matching
      - Job allocation & Reminder example (Gelernter)

→ Seamless support for…
  - Distributed data structures management
    - Partial data structures
  - All form of communication & Synchronisation

→ Basic orthogonal mechanisms that can be composed flexibly to obtain high level coordination patterns

- Some formal semantics (see Viroli seminar)

Tuple Spaces/Linda extensions

- Extending the communication language
  - XML based tuples (ex: XMLSpaces)
  - Java object tuples (ex: JavaSpaces)
  - Logic-based tuples (ex: ReSpecT)

  …
- Extending the coordination language
  - adding coordination primitives:
    - Non-blocking behaviour
      - inp, rdp
    - Inall, rdall, copy, copycollect,…
- Extending medium structure and topology
  - multiple tuple spaces (ex: TuCSoN ...)
  - nested spaces (ex: Bauhaus Linda)
- Extending medium behaviour
  - Programmable tuple spaces (ex: ReSpecT/TuCSoN, MARS…)
  - Event management (ex: JavaSpaces)

Benefits

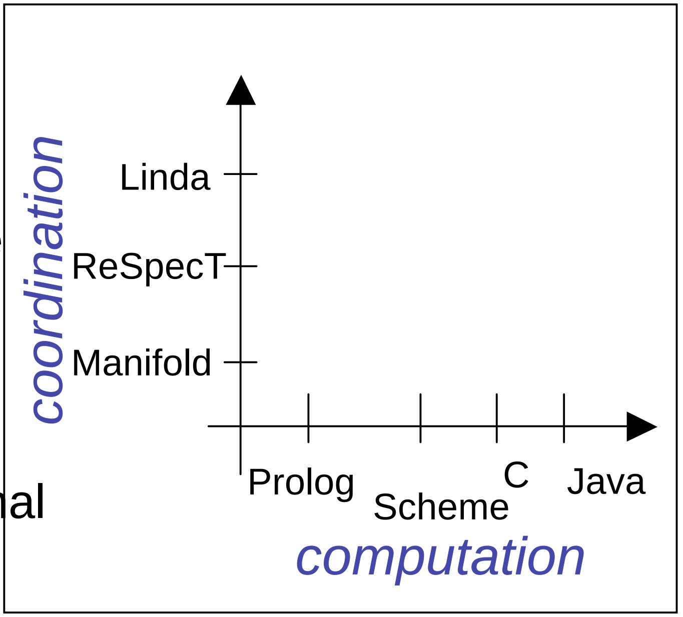
- *Ortogonality (Separation) of Coordination and Computation Languages*
  - computational languages as sort of degenerate coordination language in the form of global variables and argument passing
  - Ex: “Linda and friends”
    - Linda+C, Linda+Prolog, Linda+Fortran,…
- *Generality*
  - The same general purpose coordination language can be used in different coordination contexts, gluing different kind of computations

→ *Heterogeneity*

- gluing computation of heterogeneous computational models, all in the same coordination context

→ *Portability/Reusability*

- Reusability (recycle-ability) in reusing application, implementation, tools and heterogeneous programmer expertise in the same coordination context



PART II - SUMMING UP

- Coordination models/languages & technologies provide first class issues to model and develop coordination artifacts
  - Promotes the separation between computation and coordination/interaction issues
  - Several models with different expressivity
- An example: Tuple Space model and Linda language
  - Shared tuple spaces as coordination artifacts
  - Benefits of the generative communication

PART III

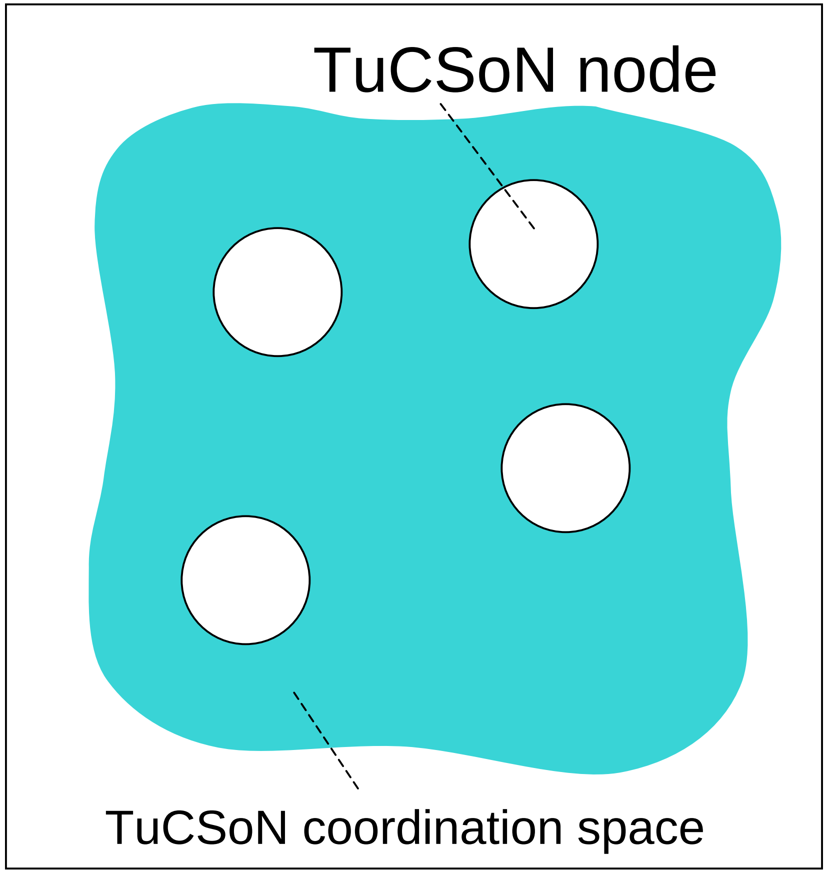
TuCSoN model and technology

TuCSoN at a glance

- Tuple Centre Spread over the Network
- From tuple spaces to *tuple centres*
  - *Programmable* tuple spaces
  - Programmable coordination artifacts!
- Tuple centres are distributed over the network, collected in *nodes*
  - Distributed coordination artifacts

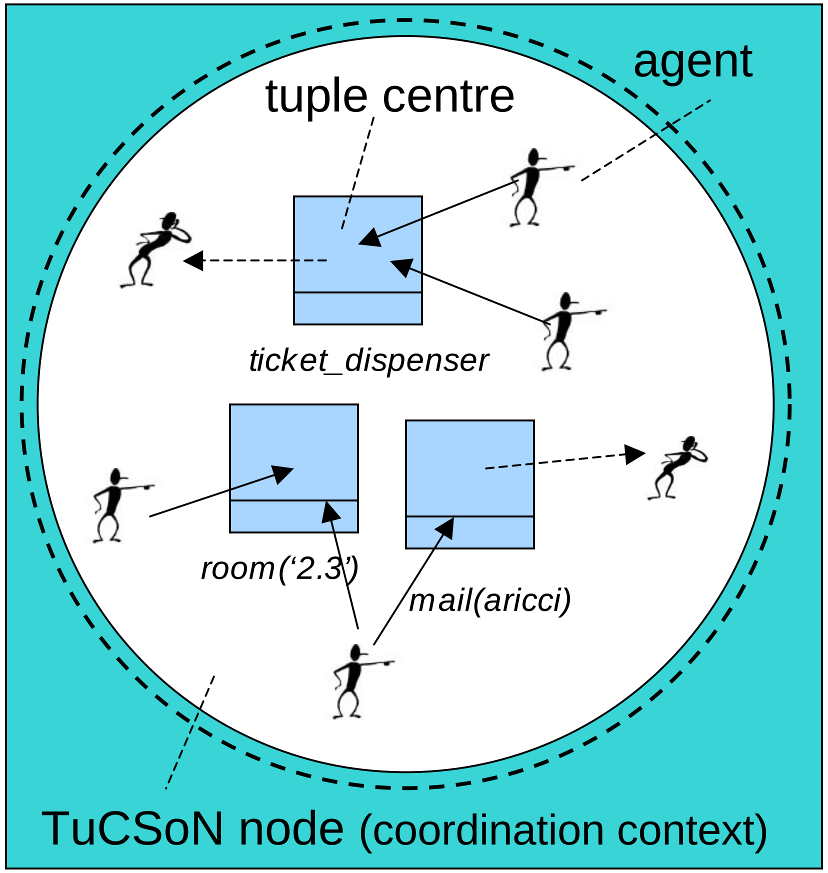
TuCSoN Coordination Space

- Coordination space = set of distributed *nodes*
  - Each TuCSoN node is an Internet node
    - Identified by the IP (logic) address



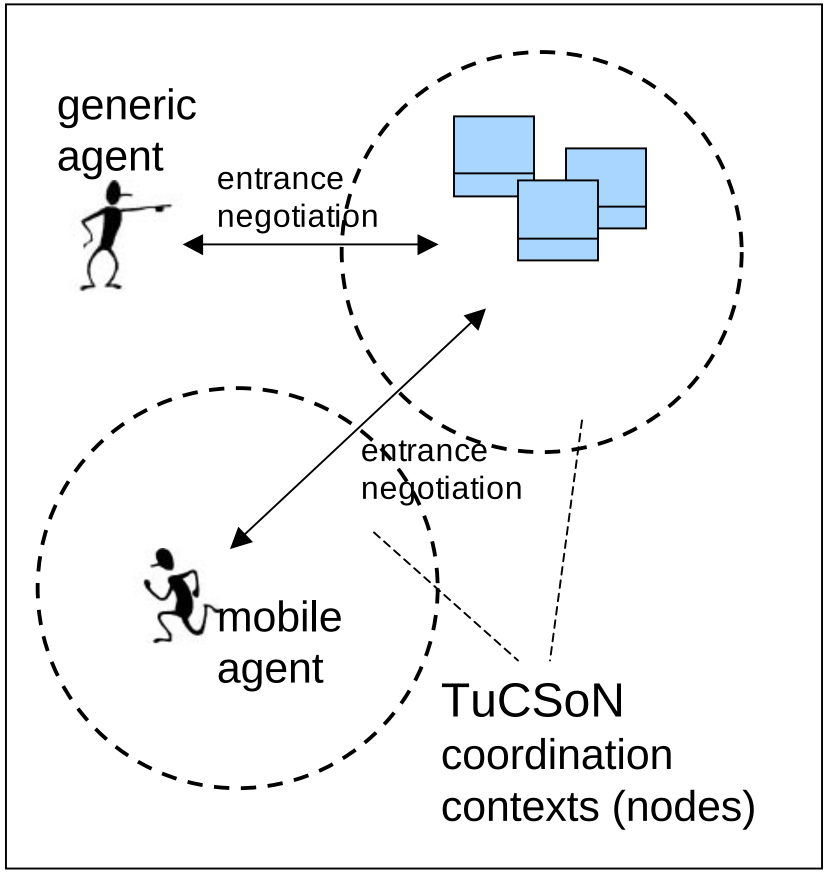
TuCSoN Node/Context

- Each TuCSoN node defines a *coordination context*, providing an open/dynamic set of *tuple centres* as coordination artifacts
  - Identified by means of a logic name (term)
    - *Ex: ticket_dispenser, mail(aricci), room('2.3'),…*
  - full tuple centre identifier: <name>@<node>
    - Ex: mail(aricci)@myhome.org, room('2.3')@ingce.unibo.it, ticket_dispenser@137.204.191.188, …



TuCSoN Node/Context

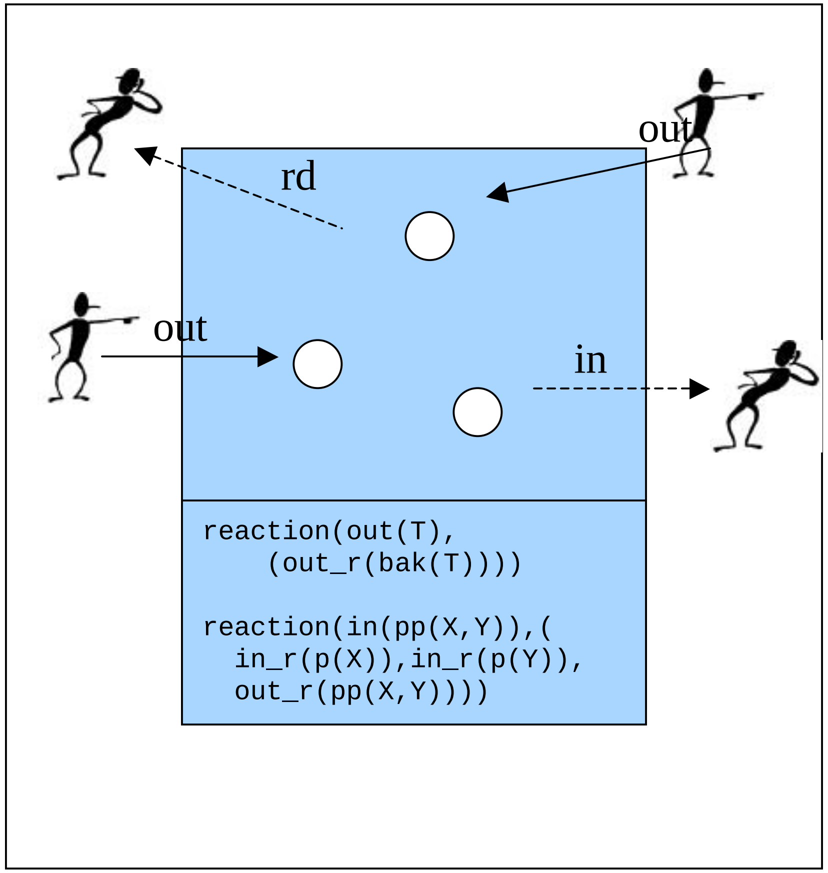
- In order to access and use the tuple centres of a node, an agent must *enter* the coordination context
  - Either logically or physically (mobile agents)



Tuple Centres

- Programmable logic tuple spaces
  - Logic tuples as communication language

- *General purpose / customizable* coordination artifacts

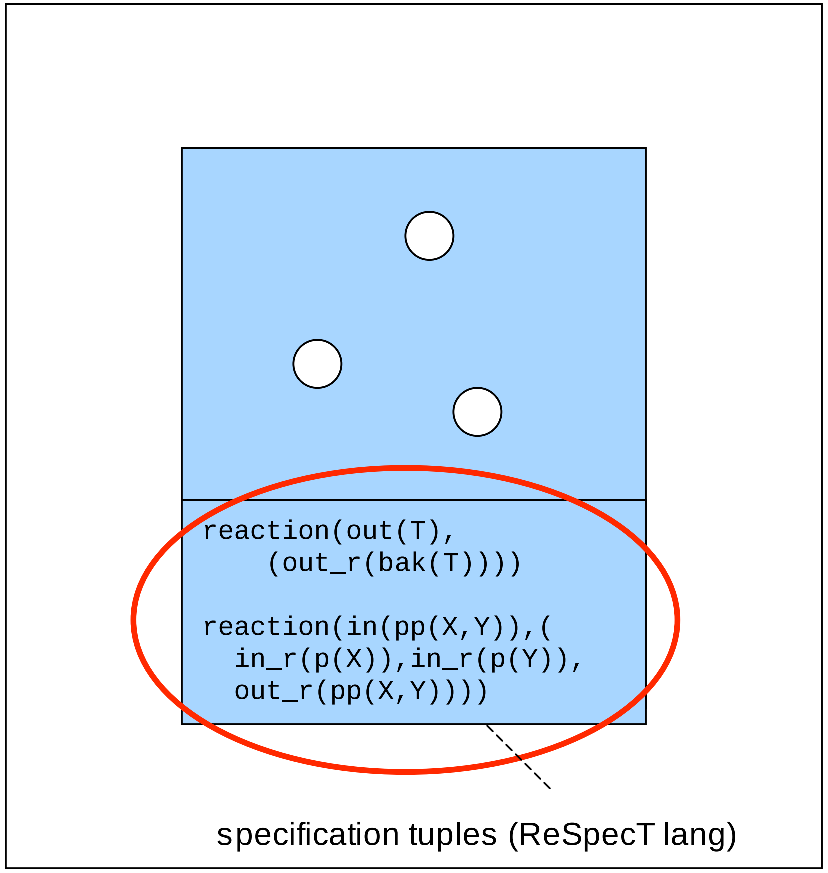


```
reaction(out(T),
    (out_r(bak(T))))

reaction(in(pp(X,Y)),(
  in_r(p(X)),in_r(p(Y)),
  out_r(pp(X,Y))))
```

Tuple Centres features

- *Programmable*
  - Tuple centre behaviour can be *programmed* to enact the desired coordination policies
  - *ReSpecT* language as programming language
    - Programs as set of logic tuples (*reactions*) specifying medium behaviour reacting to interaction events
  - [vision] tuple centres as a general purpose coordination artifacts customizable by means of the ReSpecT logic based language

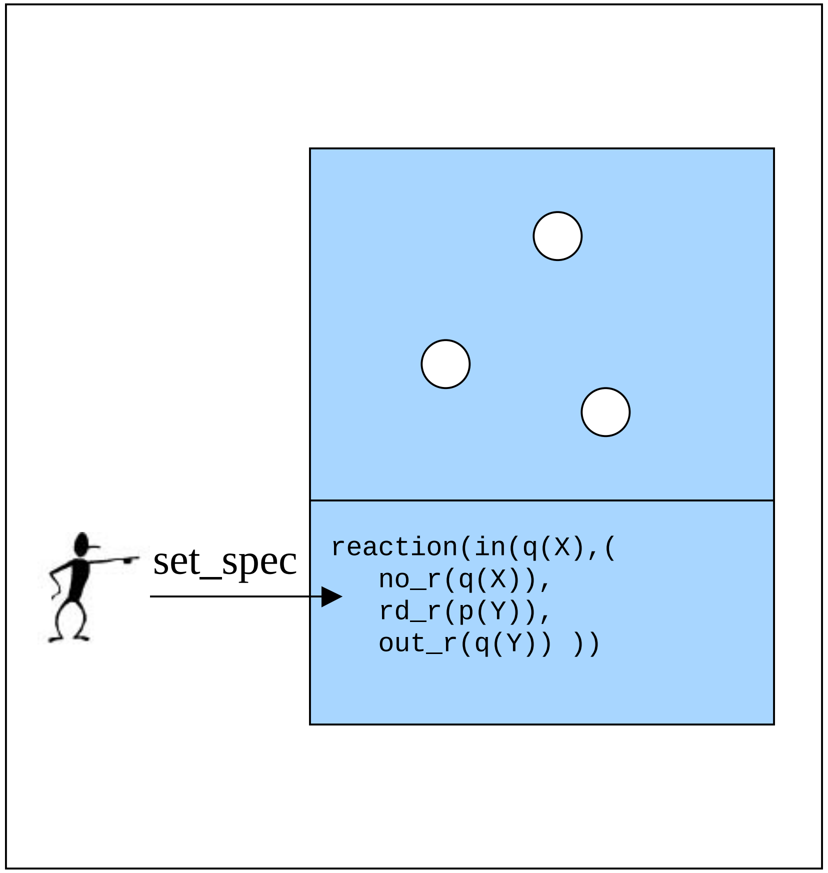


```
reaction(out(T),
    (out_r(bak(T))))

reaction(in(pp(X,Y)),(
  in_r(p(X)),in_r(p(Y)),
  out_r(pp(X,Y))))
```

Tuple Centres features

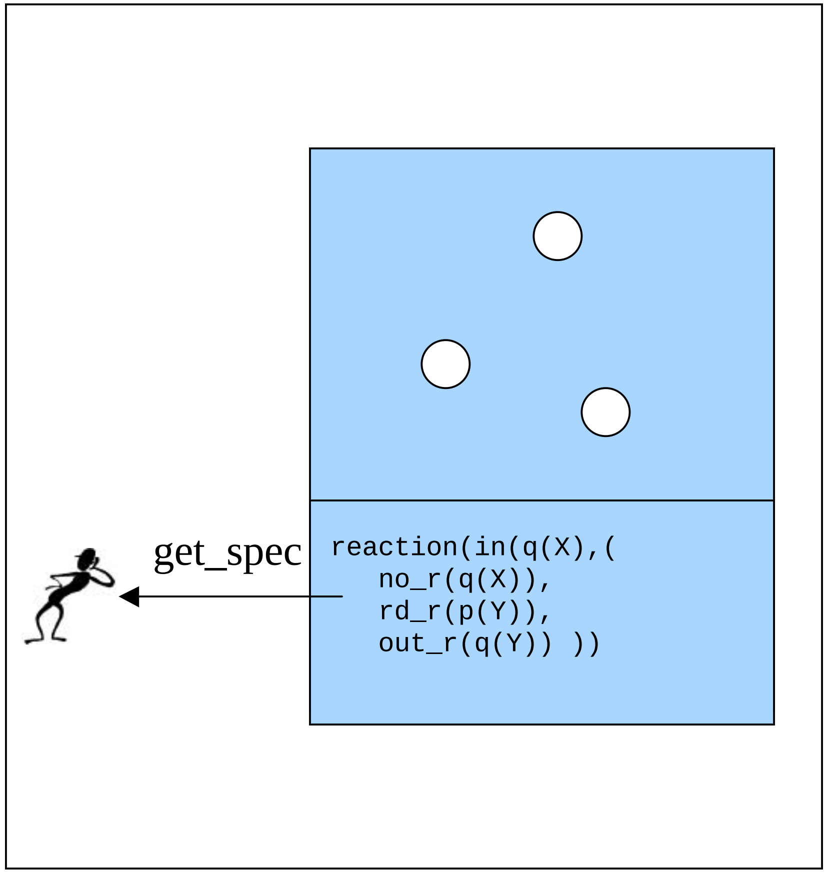
- *Adaptable at runtime*
  - Tuple centre behaviour can be changed/adapted dynamically, at runtime, by reprogramming the artifact



```
reaction(in(q(X),(
   no_r(q(X)),
   rd_r(p(Y)),
   out_r(q(Y)) ))
```

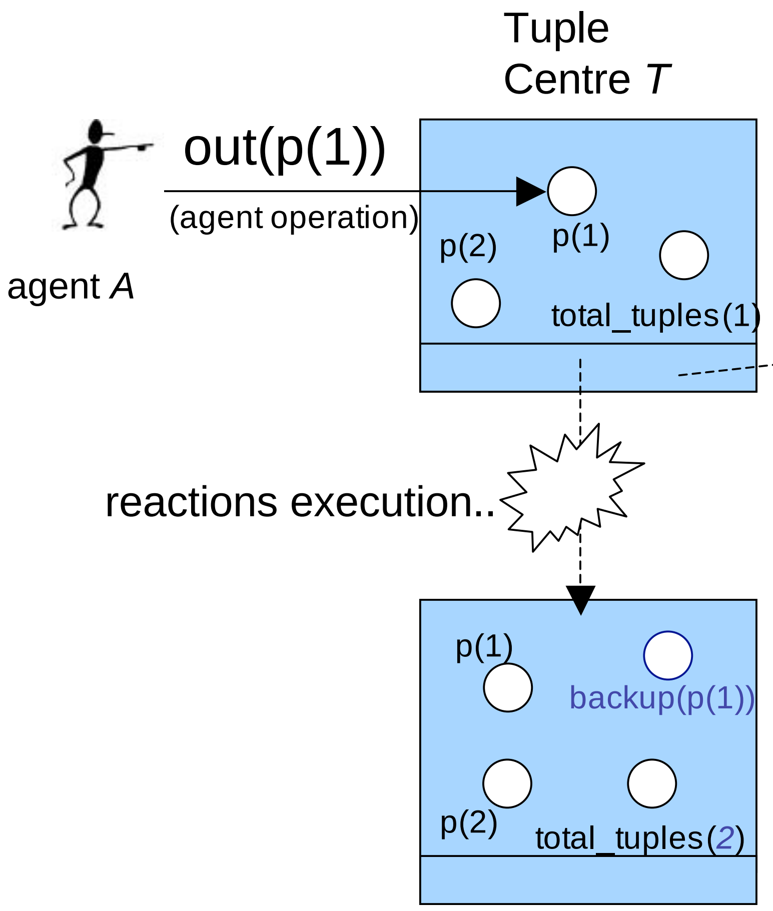
Tuple Centres features

- *Inspectable at runtime*
  - Tuple centre behaviour can be inspected dynamically, at runtime



```
reaction(in(q(X),(
   no_r(q(X)),
   rd_r(p(Y)),
   out_r(q(Y)) ))
```

Simple examples



Current behaviour of the tuple centre (pseudocode):

When a tuple T is inserted, produce a tuple `backup(T)`

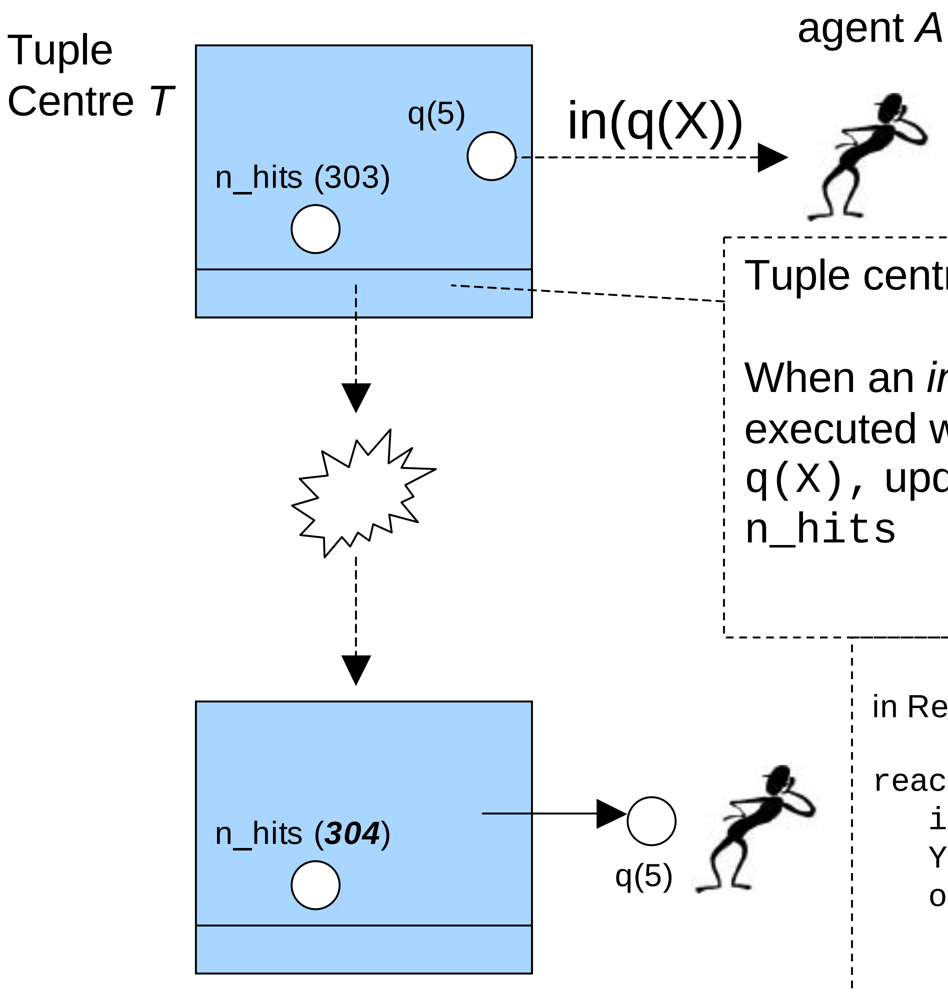
When a tuple p(X) is inserted, update the tuple `total_tuple(N)` (retrieve and store the tuple with N incremented)…

in ReSpecT:

```
reaction(out(T),(
   out_r(backup(T)))).

reaction(out(p(X)),(
   in_r(total_tuples(N)),
   N1 is N + 1,
   out_r(total_tuples(Y1)))).
```

Simple examples



Tuple centre behaviour:

When an *in* operation is executed with template `q(X)`, update the tuple `n_hits`

in ReSpecT:

```
reaction(in(q(X)),(
   in_r(n_hits(Y)),
   Y1 is Y + 1,
   out_r(n_hits(Y1)))).
```

TuCSoN Technology (1)

- TuCSoN API
  - Virtually any hosting language
    - Currently: Java, Prolog
      - Support for Java and Prolog agents
  - Heterogeneous hardware support:
    - Currently: desktop PC, PDA (Compaq iPaq, Palm)
    - In the future: LEGO, embedded computing

TuCSoN Technology (2)

- TuCSoN Service
  - Booting the TuCSoN Service daemon
    - The host becomes a TuCSoN node
    - With current version (1.3.0):

      ```
      java -cp tucson.jar alice.tucson.runtime.Node
      ```
- TuCSoN Tools
  - ***Inspectors***
    - Fundamental tool to monitor tuple centre communication and coordination state, and to debug tuple centre behaviour
    - *debugger for coordination artifacts*
      - Observing and debugging agent interaction
    - With current version (1.3.0):

      ```
      java -cp tucson.jar alice.tucson.ide.Inspector
      ```
  - TuCSoNShell
    - Shell interface for human agents
    - With current version (1.3.0):

      ```
      java -cp tucson.jar alice.tucson.ide.CLIAgent
      ```

PART III - SUMMING UP

- TuCSoN world
  - Tuple centre as general purpose and customizable coordination artifacts
  - Coordination laws and strategy are specified suitably forging the artifacts (programming the tuple centres) and establishing the agent interaction protocols in terms of exchanged tuples
- TuCSoN technology is available

PART III - NOTES (1)

TuCSoN live

<TuCSoN on the fly>

- Booting a TuCSoN node
- Using a tuple centre (as a human agent)
  - TuCSoN shell tool
- Inspecting and debugging tuple centres
  - TuCSoN inspector

<Development in TuCSoN>

- Building simple systems
  - Experiments with the “Hello world” simple Java agent
  - Creating simple coordination among Java, human and Prolog agents

TuCSoN in Java (1)

```
import alice.tucson.api.*;
import alice.logictuple.*;

public class Test {
  public static void main(String[] args) throws Exception {
    TucsonContext cn = Tucson.enterDefaultContext();
    TupleCentreId tid = new TupleCentreId("test_tc");
    cn.out(tid, new LogicTuple("p",new Value("hello world")));
    LogicTuple t=cn.in(tid, new LogicTuple("p",new Var("X"));
    System.out.println(t);
  }
}
```

TuCSoN in Java (2)

```
import logictuple.*;
import tucson.api.*;

public class Test2 {
    public static void main (String args[]) throws Exception{

        AgentId aid=new AgentId("agent-0");
        TucsonContext cn = Tucson.enterContext(new DefaultContextDescription(aid));

        // put the tuple value(1,38.5) on the temperature tuple centre
        TupleCentreId tid=new TupleCentreId("temperature");
        LogicTuple outTuple=LogicTuple.parse("value(1,38.5)");
        cn.out(tid,outTuple);

        // retrieve the tuple using value(1,X) as a template
        LogicTuple tupleTemplate=new LogicTuple("value",
                                                new Value(1),
                                                new Var("X"));
        LogicTuple inTuple=cn.in(tid,tupleTemplate);

        cn.exit();
        System.out.println(inTuple);
    }
}
```

TuCSoN in Java: A simple agent

```
import alice.tucson.api.*;
import alice.logictuple.*;

class MyAgent extends Agent {
  protected void body(){
    try {
      TupleCentreId tid = new TupleCentreId("test_tc");
      out(tid, new LogicTuple("p",new Value("hello world")));
      LogicTuple t=in(tid, new LogicTuple("p",new Var("X"));
      System.out.println(t);
    } catch (Exception ex){}
  }
}

public class Test {
  public static void main(String[] args) throws Exception {
    AgentId aid=new AgentId("alice");
    new MyAgent(aid).spawn();
  }
}
```

TuCSoN in Prolog (tuProlog)

```
:- load_library(’alice.tuprologx.lib.TucsonLibrary’).
:- solve(go).

go:-
  test_tc ? out(p(’hello world’)),
  test_tc ? in(p(X)),
  write(X), nl.
```

<TuCSoN environment overview>

- A look to the API
  - Java and Prolog
- A look to infrastructure & tools deployment
- A look to some agents

<TuCSoN Internals>

- A look to the design & development
  - “alice” open source project
    - Tuple centre framework (alice.tuplecentre)
    - tuProlog (alice.tuprolog, alice.tuprologx)
    - ReSpect (alice.respect, alice.logictuple)
    - TuCSoN (alice.tucson)

PART III - NOTES (2)

ReSpecT TUPLE CENTRE
model and language

Programmable Tuple Spaces

- Tuple Centres = Programmable Tuple Spaces
  - The behaviour of the medium in response to communication events is no longer fixed once and for all by the model, but can be defined according to the required coordination policies
    - Coordination laws no longer fixed, but specified/programmed according to the coordination need
  - The medium behaviour is enriched in terms of state transitions (*reactions*) performed in response to the occurrence of standard communication events (ex: insertion of a tuple, retrieve of a tuple,…)
  - Tuple centres as general purpose *reactive* associate blackboards
- Same standard tuple space interface…
  - entities perceive the tuple centre as a standard tuple space
- …but can behave in a very different way with respect to a tuple space, since its behaviour can be specified so as to encapsulate the coordination rules governing the interaction

Tuple Centre behaviour: reactions

- More formally, tuple centres enhance tuple spaces with with *behaviour specification,* defining tuple centre behaviour in response to communication events
  - Communication events examples: the tuple T has been inserted in the space, an *in* operation been beformed with template TT,…
- Behaviour specification is expressed in terms of a *reaction specification language*, and associates any communication event possibly occurring in the tuple centre to a (possibily) empty set of computational activities called *reactions*
- Reactions act on the communication/coordination state
  - Can access and modify the current tuple centre state
    - adding, removing, reading tuples…
  - Can access all the information related to the triggering communication event
    - The operation related generating communication events, the entity identity performing the operations,…

Tuple Centre dynamics

- Multiple reactions in one shot
  - Each communication event may trigger a multeplicity of reactions which are executed locally to the tuple centre
- Super-imposing tuple space behaviour
  - When a communication event occurs, a tuple centre first behaves like a standard tuple space, then executes all the triggered reactions before serving any other entity-triggered communication event and any other coordination primitives invocation
  - A tuple centre with empty behaviour = a tuple space
- Atomicity
  - The observable behaviour of a tuple centre in response to a communication event is still perceived by coordinable as a single-step/atomic state transition of the medium, as in the case of tuple spaces
    - Reactions are not observable by coordinables

The ReSpecT language

- ReSpecT is a language for the specification of the behaviour of tuple centres
  - Logic tuples as communication language
    - based of first-order logic, where a tuple is a fact
    - *Unification* as tuple matching mechanism
      - Examples: p(1,_) and p(1,2) match, p(X,X,1) and p(1,Y,Z) match, p(X,X) and p(1,2) don’t match…
- Reactions defined through logic tuples too
  - A *specification tuple* **`reaction(Op,R)`** associates the event generated by the incoming communication operation Op to the reaction R. Example:

    reaction(out(p(1)), …)
  - A reaction is defined a sequence of *reaction goals*, which may access properties of the occurred communication event, perform simple term operations, manipulate tuples in the tuple centre. Examples:

```
reaction(out(T), ( out_r(backup(T)) )).
reaction(out(p(X)), ( in_r(total_tuples(N)), N1 is N + 1,
                      out_r(total_tuples(N1)) )).
reaction(in(q(X)), ( no_r(q(_)), out_r(q(5)) )).
```

ReSpecT primitives

Main ReSpecT predicates for reactions

Tuple space access and modification	
`out_r(T)`	succeeds and inserts tuple `T` into the tuple centre
`rd_r(TT)`	succeeds, if a tuple `T` matching template `TT` is found in the tuple centre, by unifying `T` with `TT`; fails otherwise
`in_r(TT)`	succeeds, if a tuple `T` matching template `TT` is found in the tuple centre, by unifying `T` with `TT` and removing `T` from the tuple centre; fails otherwise
`no_r(TT)`	succeeds, if no tuple matching template `TT` is found in the tuple centre; fails otherwise
Communication event information	
`current_tuple(T)`	succeeds, if `T` matches the tuple involved by the current communication event
`current_agent(A)`	succeeds, if `A` matches the identifier of the agent which triggered the current communication event
`current_op(Op)`	succeeds, if `Op` matches the operation which triggered the current communication event
`current_tc(N)`	succeeds, if `N` matches the identifier of the tuple centre where the reaction is executed
`pre`	succeeds in the *pre* phase of any operation
`post`	succeeds in the *post* phase of any operation
`success`	succeeds in the *pre* phase of any operation, and in the *post* phase of any successful operation
`failure`	succeeds in the *post* phase of any failed operation

The ReSpecT language

- Reaction goals are executed sequentially
  - Reaction goals can trigger new reactions
    - Reacting on out_r, in_r, rd_r, no_r primitives..
- Success/failure semantics of each reaction execution
  - A reaction as a whole is either a *successful* or *failed reaction* if *all* its reaction goals succeed or not
  - *Transactional semantics*
    - a successfull reaction can atomically modify the tuple centre state, a failed yelds no results at all
- The execution order of (possibly) multiple triggered reactions is not deterministic
- All the reactions triggered by a given communication event are executed before serving any other communication event
  - Coordinables perceive only the final result of the execution of the communication event *and* the set of all the triggered reactions

Facts about ReSpecT

- Turing equivalent language
  - Powerful enough to express any computation/algorithm acting on the interaction space
  - → General purpose enaugh to support the specification of any compuatble coordination policies
  - → Expressivity issues (see Viroli seminar)
- Formal semantics (see Viroli seminar)
  - Fundamental to understand coordination activities
  - The basis for supporting formal analysis and reasoning about interaction dynamics

PART III - NOTES (2)

Some coordination patterns

developed using

Linda and TuCSoN

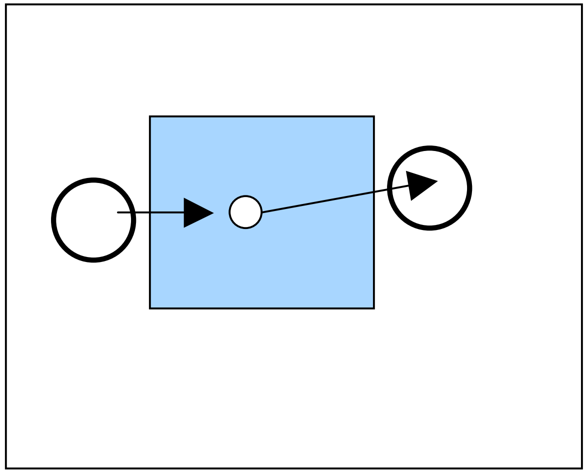
Some coordination patterns in Linda & TuCSoN (ReSpecT)

- Basic coordination
  - Communication & Interoperability
    - Managing information flow
  - Basic Synchronisation
    - Managing temporal dependencies
  - Basic Resource sharing / allocation
    - Task allocation

- More articulated coordination
  - Workflow Management
  - Transactions
  - Event-based Patterns
    - Notifications
    - Publish/Subscribe …

Enabling communication (1)

- Message Passing



SENDER:

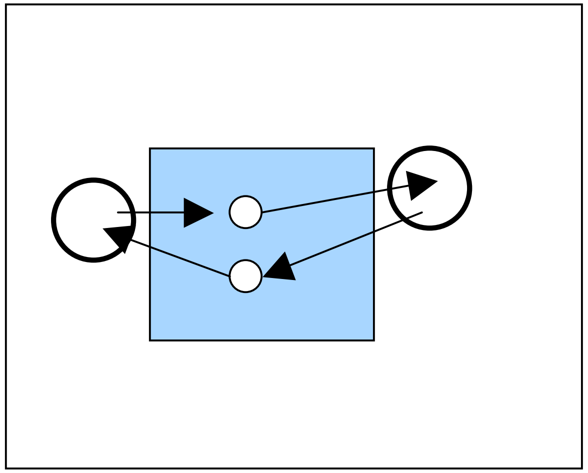
```
out(msg(agentB,content(‘test’,13)))
```

RECEIVER (called agentB):

```
in(msg(agentB,Info))
```

Enabling communication (2)

- RPC style



SERVICE USER:

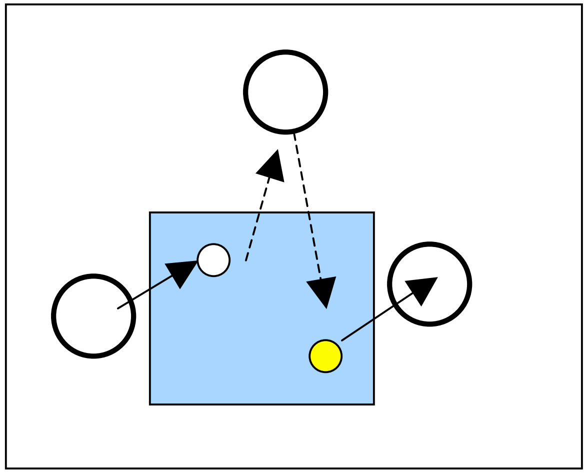
```
…
out(compute_sum(5,8,me))
in(compute_sum_result(me,Value)
…
```

SERVICE PROVIDER

```
in(compute_sum(X,Y,Who))
Sum ← X + Y
out(compute_sum_result(Who,Sum))
```

Enabling interoperability

- Mediating different ontologies



Good, *but*
-the mediation as a coordination activity is charged upon an entity (the service mediator), not upon the medium
(Conceptual mismatch → engineering drawbacks)

SERVICE USER:

```
…
out(compute_sum(5,8,me))
in(compute_sum_result(me,Value)
…
```

SERVICE PROVIDER

```
in(make_sum(term(X,Y)))
Sum ← X + Y
out(sum_result(X,Y,Sum))
```

SERVICE MEDIATOR

```
in(compute_sum(X,Y,Who))
out(service_requested(sum(X,Y),Who))
```

```
in(sum_result(X,Y,Sum))
in(service_requested(sum(X,Y),Who))
out(compute_sum_result(Who,Sum))
```

Interoperability in TuCSoN

- Ontology mediation charged upon the medium

Linda Style

SERVICE USER:

```
…
out(compute_sum(5,8,me))
in(compute_sum_result(me,Value)
…
```

SERVICE PROVIDER

```
in(make_sum(term(X,Y)))
Sum ← X + Y
out(sum_result(X,Y,Sum))
```

SERVICE MEDIATOR

```
in(compute_sum(X,Y,Who))
out(service_requested(sum(X,Y),Who))


in(sum_result(X,Y,Sum))
in(service_requested(sum(X,Y),Who))
uut(compute_sum_result(Who,Sum))
```

⇒

TuCSoN Style

SERVICE USER:

```
…
out(compute_sum(5,8,me))
in(compute_sum_result(me,Value)
…
```

SERVICE PROVIDER

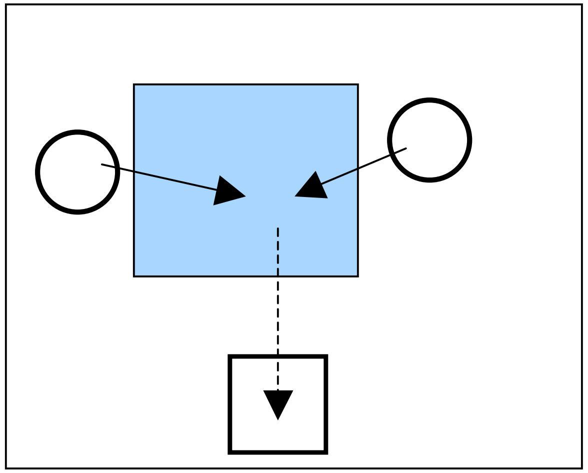
```
in(make_sum(term(X,Y)))
Sum ← X + Y
out(sum_result(X,Y,Sum))
```

MEDIATION POLICY in ReSpecT

```
reaction(out(compute_sum(X,Y,Who)),(
    in_r(compute_sum(X,Y,Who)),
    out_r(service_requested(sum(X,Y),Who)),
    out_r(make_sum(term(X,Y)) )).
reaction(out(sum_result(X,Y,Sum)),(
    in_r(sum_result(X,Y,Sum)),
    in_r(service_requested(sum(X,Y),Who)),
    out_r(compute_sum_result(Who,Sum)) )).
```

Basic synchronisation (1)

- Synchronisation



Synchronised agent:

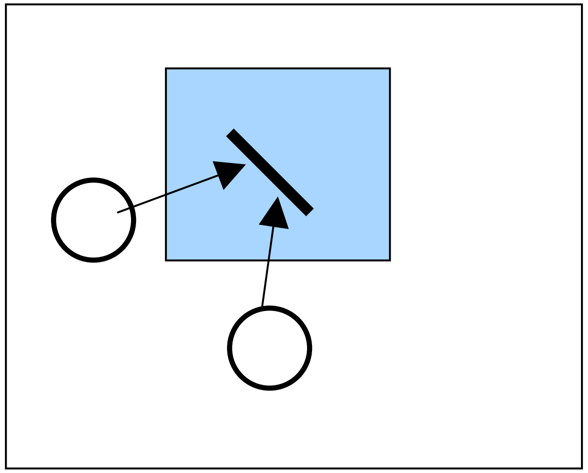
```
<outside sync region>
…
in(token)
<inside sync region>
out(token)
…
<outside sync region>
…
```

HYPOTHESIS:
Initial space content with the tuple `token`

To have synchronised region allowing N users inside
→ N tuples `token`

Basic synchronisation (2)

- Barrier Synchronisation



Agent A:

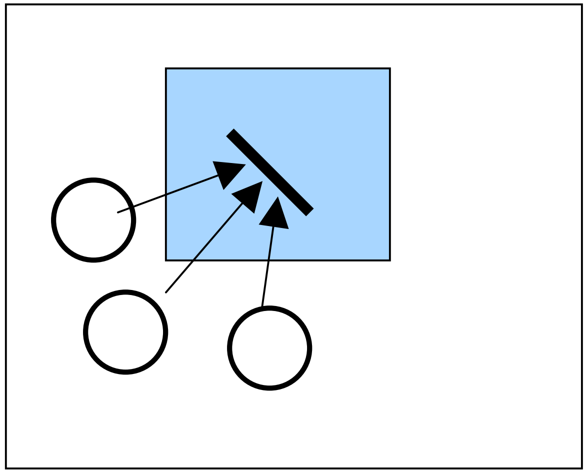
```
...
<before barrier>
...
out(reay(agentA))
rd(ready(agentB))
<agents A and  B
are now synchronised>
```

Agent B:

```
...
<before barrier>
...
out(ready(agentB))
rd(ready(agentA))
<agents B and  A
are now synchronised>
```

Basic synchronisation (3)

- Barrier Synchronisation with 3+ entities



Agent A:

```
…
out(ready(agentA))
rd(ready(agentB))
rd(ready(agentC))
…
```

Agent B:

```
…
out(ready(agentB))
rd(ready(agentA))
rd(ready(agentC))
…
```

Agent C:

```
…
out(ready(agentC))
rd(ready(agentA))
rd(ready(agentB))
…
```

Good, *but*

-Adding an agent → changing the behaviour of all the other agents

-Every agent must be aware of all the other ones

Barrier synchronisation in TuCSoN

- Encapsulating the barrier synchronisation policy

Linda Style

Agent A:
```
…
out(ready(agentA))
rd(ready(agentB))
rd(ready(agentC))
…
```

Agent B:
```
…
out(ready(agentB))
rd(ready(agentA))
rd(ready(agentC))
…
```

Agent C:
```
…
out(ready(agentC))
rd(ready(agentA))
rd(ready(agentB))
…
```

TuCSoN Style

ANY agent:
```
…
out(ready)
rd(ready_all)
…
```

BASIC BARRIER SYNCHRONISATION in ReSpecT

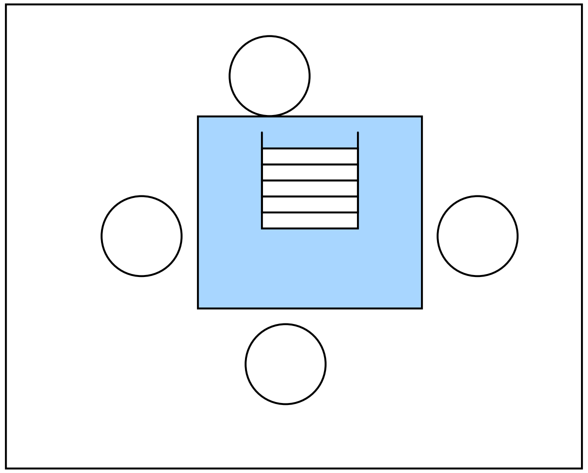
```
reaction(out(ready),( in_r(ready),
    in_r(ready_entities(N)), N1 is N+1,
    out_r(ready_entities(N1)) )).

reaction(out_r(ready_entities(N)),(
    rd_r(barrier_size(N)), out_r(ready_all))).
```

HYPOTHESIS:
- Initial space content with `ready_entities(0)`
- `barrier_size(N)` tuple to specify number of coordinables to be synchronised

Resource sharing/allocation

- A dynamic/open set of agents accessing the same resource (ex: a printer) according to a coordination policy (ex: First Come First Served)



Each user agent:

```
…
in(next_ticket(T))
T1 ← T + 1
out(next_ticket(T1))
in(turn(T))
  <use the resource>
out(turn(T1)
…
```

HYPOTHESIS: Initial space content includes the tuples:

```
next_ticket(0)
turn(0)
```

Good, *but*

-Changing the coordination policy

→ changing all the other entities

-Malicious/Failing agents?

Resource sharing/allocation in TuCSoN (I)

- Encapsulating the sharing policy
  - Scale down complexity to a synchronisation problem

Linda Style

Each user agent:

```
…
in(next_ticket(T))
T1 ← T + 1
out(next_ticket(T1))
in(turn(T))
  <use the resource>
out(turn(T1)
…
```

TuCSoN Style

EACH USER:

```
…
in(resource_token(<my name>))
<use the resource>
out(resource_token(<my name>))
…
```

SHARING COORDINATION LAWS in ReSpecT:

```
reaction(in(resource_token(Who)),( pre,
  in_r(tickets(N)), N1 is N + 1,
  out_r(tickets(N1)),
  out_r(turn(Who,N)) )).

reaction(out_r(turn(Who,N)),(
  rd_r(current_turn(N)),
  out_r(resource_token(Who)) )).

reaction(out(resource_token(Who)),(
  in_r(resource_token(Who)),in_r(turn(Who,N)),
  in_r(current_turn(N)), N1 is N+1,
  out_r(current_turn(N1)) )).
```

Resource sharing/allocation in TuCSoN (II)

- Changing/adapting the sharing policy
  - From FIFO strategy to LIFO strategy

unchanged aehaviour for agents

Each user agent:

```
…
in(resource_token(<my name>))
<use the resource>
out(resource_token(<my name>))
…
```

changing only the glue code

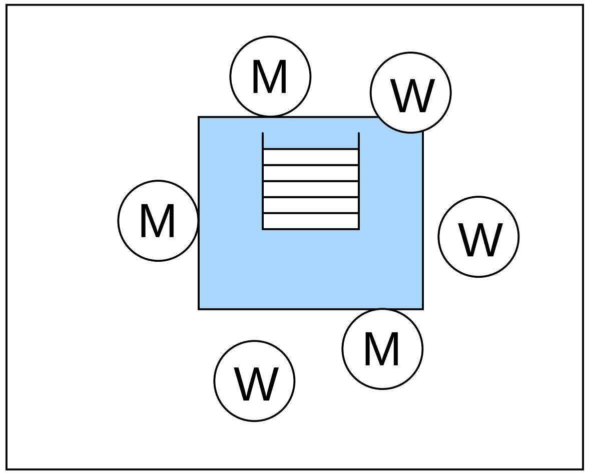
LIFO SHARING POLICY:

```
reaction(in(resource_token(Who)),( pre,
  in_r(last(N)), N1 is N + 1,
  out_r(last(N1)),
  out_r(heap(Who,N1)),
  out_r(check) )).

…
```

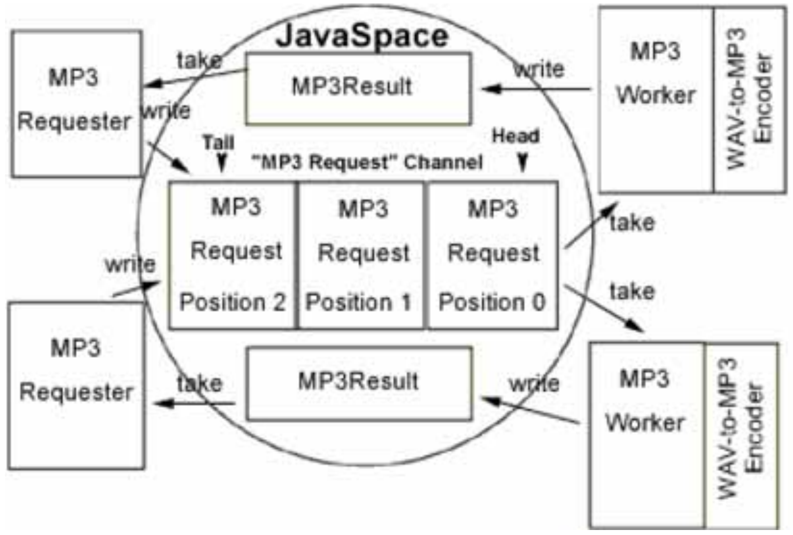
Task allocation

- Task allocation to an open set of *workers*, with task request provided by an open set of *masters,* according to some policy
  - MP3 Service Case Study: building a distributed Internet-based MP3 encoding service
    - masters request WAV → MP3 conversion
    - workers provide the conversion
    - service provision policy: FIFO
      - possibly dynamically/adaptable



from the articles "*Make Room for JavaSpaces*" by Susan Hupfer – Java World electronic magazine, Jiniology Serie

Also in the book:
"Java Spaces: Principle and Patterns" AW.



Task allocation: Linda approach

MP3 REQUESTER (master)

```
while (true) {
    acquireFromGUI(FileName)
    readRawData(FileName,RawData)
    in(tail(T))
    T1 ← T + 1
    out(tail(T1))
    out(mp3request(T1,FileName,
                   RawData,myId))
    in(mp3result(FileName,
                 ResultData,myId))
}
```

MP3 CONVERTER (worker)

```
while (true) {
    rd(tail(T))
    in(head(H))
    if (T<H){
        out(head(H))
    } else {
         H1 ← H + 1
       out(head(H1))
       in(mp3request(H,FileName,Data,
                     FromWho))
       MP3Data ← from_raw_to_data(Data)
       out(mp3result(FileName,
                     MP3Data,FromWho))
    }
}
```

good, but the coordination burden is almost upon the coordinables
-changing policy → changing coordinables
-...

Task allocation: TuCSoN approach

MP3 REQUESTER (master)

```
while (true) {
    acquireFromGUI(FileName)
    readRawData(FileName,RawData)
    out(mp3request(FileName,RawData,myId))
    in(mp3result(FileName,ResultData,myId))
}
```

MP3 REQUESTER (worker)

```
while (true) {
    in(mp3request(FileName,Data,FromWho))
    MP3Data ← from_raw_to_data(Data)
    out(mp3result(FileName,MP3Data,FromWho))
}
```

FIFO TASK ALLOCATION POLICY in ReSpecT

```
reaction(out(request(_,_,_)),(
  rd_r(workers_available(N)),
  N>0)).
reaction(out(request(Name,Data,From)),(
  rd_r(workers_available(N)),
  N == 0,
  in_r(tail(T)), T1 is T + 1, out_r(tail(T1)),
  in_r(request(Name,Data,From)),
  out_r(req_queue(T1,Name,Data,From)))).
reaction(in(request(Name,Data,From)),( pre,
  rd_r(head(H)),rd_r(tail(T)),
  T < H)).
```

```
reaction(in(request(Name,Data,From)),( pre,
  in_r(head(H)), rd_r(tail(T)),
  T >= H,
  H1 is H + 1, out_r(head(H1)),
  in_r(req_queue(H,N1,D1,F1)),
  out_r(request(N1,D1,F1)))).
reaction(in(request(_,_,_)),( pre,
  in_r(workers_available(N)),
  N1 is N + 1,out_r(workers_available(N1)))).
reaction(in(request(_,_,_)),( post,
  in_r(workers_available(N)),
  N1 is N - 1, out_r(workers_available(N1)))).
```

PART IV

Using TuCSoN for your projects

[discussion]

APPENDIX

Selected Bibliography & References

Selected bibliography (1)

- Interaction
  - Why Interaction is more powerful that algorithms (Wegner) – Communication of ACM, Vol. 40, No. 5, May 1997
  - Interactive Foundation of Computing (Wegner) – Theoretical Computer Science, Vol. 192, No. 2, February 1998
- Coordination Overview & Surveys
    - Coordination Languages and their Significance (Gelernter, Carriero) – Communication of ACM, Vol. 33, No. 2, February 1992
    - Coordination Models and Languages as Software Integrators (Ciancarini) – ACM Computing Surveys, Vol. 28, No. 2, June 1996
    - Programmable Coordination Media (Denti, Natali, Omicini) – 2nd International Conference (COORDINATION '97), Proceedings, LNCS 1282, Springer-Verlag, 1997
    - Coordination Models and Languages (Arbab, Papadopoulos) – Advances in Computers, Vol. 46, Academic Press, August 1998
    - The Interdisciplinary Study of Coordination (Malone, Crostow) – ACM Computing Surveys, Vol. 26, No. 1, March 1994

Selected bibliography (2)

- Coordination Models/Languages/Infrastructures and TuCSoN
  - Tuple Spaces & Linda
    - Generative Communication in Linda (Gelernter) – ACM Transactions on Programming Languages and Systems, Vol. 7, No. 1, January 1985
  - Tuple Centre & ReSpecT
    - On the Expressive Power of a Language for Programming Coordination Media (Denti, Natali, Omicini), 1998 ACM Symposium on Applied Computing (SAC), 1998
    - From Tuple Spaces to Tuple Centres (Omicini, Denti) – Science of Computer Programming, No. 41, 2001
    - Formal ReSpecT (Omicini, Denti) – Electronic Notes in Theoretical Computer Science, No. 48, 2001
  - TuCSoN
    - <http://www.lia.deis.unibo.it/Staff/AndreaOmicini publication section>
    - Several other works can be found at the articles page of the web sites:
      - http://www.lia.deis.unibo.it/Staff/EnricoDenti
      - http://www.lia.deis.unibo.it/Staff/AndreaOmicini
      - http://www.lia.deis.unibo.it/Staff/AlessandroRicci

References

my email address: aricci@deis.unibo.it

- For questions, explorations, ideas, articles & technology requests
  - Development issues
    - using Java, Prolog, C, C++, C# with TuCSoN
- For thesis about coordination technologies
  - Porting TuCSoN everywhere :)
  - Infrastructure and System development issues
    - Representing and Coordinating real world services in the TuCSoN world (CSCW services (email, FTP,…), Web Services, …)
    - Pervasive computing and Intelligent environments
      - accessing to TuCSoN coordination contexts from mobile devices (PDA, Cellular phones, LEGO)

(DEIS research group:

Enrico Denti, Gianluca Moro, Antonio Natali, Andrea Omicini,
Alessandro Ricci, Mirko Viroli)

Among the available thesis...

- Context: Coordination for pervasive computing and intelligent environments
  - *TuVoC*
    - Design & development of a vocal service for TuCSoN coordination contexts
  - *TuQu*
    - Design & development of a SQL service for TuCSoN coordination contexts
  - *TuCStorm*
    - Porting TuCSoN on LEGO RCX

<end>