Coordination Models, Languages and Infrastructures

Enrico Denti
enrico.denti@unibo.it
ALMA MATER STUDIORUM – Università di Bologna
Scuola di Dottorato in Ingegneria e Scienza dell'Informazione
2 luglio 2010

Outline

- The paradigm shift: from Computation to Interaction to Coordination
  - Recognising, Enabling, Governing Interaction
- Coordination Models and Languages
  - Concepts & Classification
- Tuple-based Coordination
  - From Tuple Spaces (& Linda) to Tuple Centres: ReSpecT
  - Coordination patterns in ReSpecT
- A Tuple-based Coordination Infrastructure: TuCSoN
  - Organisation, Topology, Security aspects...
  - Tools
The Paradigm Shift (1)

- The origins: a world of computation
  - main issues: computability, complexity
  - Turing Machines ('30), Church's computable functions, Random Access Machines (Shepherdson/Sturgis, '60), Recursive Functions (Kleene, '30), Lambda Calculus (Church, '30), Markov Algorithms ('60), …
  - Algol, Fortran, Lisp ('60), Basic, C, COBOL ('70), Prolog ('80), …
  - computation in the foreground
  - interaction.. what? ("just read my data, compute, and print – oh yeah")

- but the world changes..
  - from sequential (single flow of control) to concurrent & parallel computing
    - multiple control flows – where? how?
    - new concepts (processes), new issues (competition, synchronisation...)…
  - distributed computing
    - "computing is everywhere" → ok.. where? → focus on environment
    - heterogeneity of execution contexts → new metaphors (tasks, activities..)
  - greater complexity, much less formalisability

The Paradigm Shift (2)

- The origins: a world of computation
  - main issues: computability, complexity
  - Turing Machines ('30), Church's computable functions, Random Access Machines (Shepherdson/Sturgis, '60), Recursive Functions (Kleene, '30), Lambda Calculus (Church, '30), Markov Algorithms ('60), …
  - Algol, Fortran, Lisp ('60), Basic, C, COBOL ('70), Prolog ('80), …
  - computation in the foreground
  - interaction.. what? ("just read my data, compute, and print – oh yeah")

- but the world changes..
  - interaction more and more in the foreground
    - computation makes a back step
    - NOT a dimension that can be "added on top" an existing design/system
  - what does to interact means?
    - WHO interacts? Which metaphors & tools for?
    - HOW is this aspect understood and governed?
    - HOW is the system designed and engineered?
From Computation to Interaction

- Interaction ::= *a new dimension*
  - orthogonal w.r.t computation
  - broader space of concepts: time, space, communication, …
  - key role of *observation*
  - need for adequate/new Models, Methodologies, *Languages, Infrastructures, Support Tools*, …
  - but also a source of crisis..
    - the behaviour of a software component can no longer be described just as "how it computes" or "which interface it exposes"
    - unpredictability: behaviour often known only *a posteriori*
      - description in terms of interaction histories
  - a dimension to shape and understand
    - new meta-models: Persistent Turing Machine, Interactive Turing Machine (Wegner), process calculi (Pi-calculus,…),…
    - new models: transition-systems, process algebra, chemical abstract machines…

From Interaction to Coordination

- If *interaction* is a new dimension, *who/how to govern it?*

**COORDINATION ::=**

- *merging separate activities into an ensemble* (Gelernter, 1985-92)
  - Computation model → building a single computation activity
  - Coordination model → the glue that binds separate activities into an ensemble

- *managing/constraining interactions* (Wegner, 1997)

- *managing dependencies among activities* (Malone et al., 1994)
From Interaction to Coordination: Dealing with Systems

- **Coordination** as *modeling / shaping / engineering the interaction space*
- **System** view goes *beyond the composition perspective*
  - a system is *more than the sum of its parts* (non-compositionality)
    - \( \text{behaviour}(P \mid Q) = \text{behaviour}(P) + \text{behaviour}(Q) + \text{interaction}(P,Q) \)
  - a system is more than its structure
  - need for *(dynamic) glue* (structures, processes, patterns...)
- More and more emphasis on the *social aspect* of interaction
  - rather than just "the set of individuals"

---

The Paradigm Shift

**evolution of the network technology**


- functional abstractions
- algorithms
- procedure oriented languages
- derivations (proofs), compositional behavior
- closed systems, programming in the small
- logic and search in AI, formal methods
- functional requirements
- input/output transformation
- algorithmic systems
- WIFS, induction, algebras, r-points
- Turing machines, lambda-calculus
- classic logics
- state (persistence, time) abstractions
- communication (interaction), mobility
- observable behavior
- object-oriented programming
- bisimulations, emergent behavior
- open systems, programming in the large
- agent-oriented (distributed) AI, connectionist
- non functional requirements
- services over time
- reactive (persistent) systems
- NWIFS, coordination, coalgebras, r-points
- IMs (QTM, SIM, MIM, IAM, ASM...), p-calculus
- modal, temporal logics
- events, reactions, partial specifications
Coordination as a *Glue*

(Dynamic) construction of applications by gluing "LEGO-like" components, with specific interfaces and non-functional requirements, which must be adapted by means of flexible glue.

Coordination of dynamic/open set of services offered by agents/components which adhere to specific contracts possibly according to their role.

Coordination of autonomous activities, concurrent, asynchronous, with both private goals (which compete and need to be rules) and collective goals.

Coordination Everywhere (inter-disciplinary studies)

- **Computer Science**
  - Software engineering
    - Architecture Description Languages (ADLs..)
    - Component Based Frameworks (EJB,..)
    - Agent-Oriented Software Engineering (AOSE)
  - Languages
    - Aspect Oriented Programming
    - Scripting Languages
  - Concurrent (Distributed + Parallel) systems
    - Network Computing / Embedded Internet / Pervasive Computing
    - Service-Oriented Computing
      - .NET Orchestration, C# choreography...
  - (Distributed) Artificial Intelligence
    - Agent societies
- **Control/System theory**
- **Economy, Psychology and Sociology**
  - Social systems, Organisation theories,…
A Coordination Metamodel (Ciancarini, 1996)

A coordination medium which enables & governs & promotes interactions among the interacting entities (coordinables) based on/enforcing some coordination laws - enacted by the medium behaviour - defining coordination semantics

The coordinables

The glue

---

Coordination Metamodel & Languages

- **The Coordinables**
  - Entities whose mutual interaction is ruled by the model (processes, threads, objects, users, agents...)
  - focus NOT on their inner model and functioning, but on their observable behaviour

- **The Coordination Media**
  - Abstractions enabling and ruling interactions among the coordinables (semaphors, monitors, channels, tuple spaces, blackboards, pipes..)

- **The Coordination Laws**
  - Laws defining the behaviour of the coordination media (which interactions can occur, what they do / how they behave, and which information they convey with them)

From the language viewpoint, the above distinction highlights three categories of languages:

- **Computation Language** to express their inner computation
- **Communication Language** to express their concepts to others
- **Coordination Language** to perform acts of communication
In the case of Agents..

- The coordination model can be seen as an *agent interaction framework*
  - "A coordination model provides a framework in which the interaction of active and independent entities called agents can be expressed" [Ciancarini, 1996]

- What should a coordination model provide for?
  - "A coordination model should cover the issues of creation and destruction of agents, communication among agents, spatial distribution of agents, synchronization, and distribution of their actions over time" [Ciancarini, 1996]

Computation & Coordination Orthogonality

- **Coordination is orthogonal w.r.t. computation**
  - the computation language is conceptually independent w.r.t. the coordination & communication languages
  - multiple mixes are possible: a computation language can be coupled with different communication & coordination languages, and vice-versa

- **The communication language**
  - defines the phrases used by coordinables to talk & understand each other
  - defines the syntax for expressing and exchanging "communication data" (tuples, XML pages, logic tuples, (Java) objects, …)

- **The coordination language**
  - defines the interaction primitives used by coordinables to interact (Linda-like in/out/rd, send/receive (channels), push/pull (pipes…)
  - ..and obviously their semantics
Computation & Coordination Orthogonality: the Language Viewpoint (2)

- **Some notable examples**
  - just under your nose ☺: the C/C++ programming languages
    - the C/C++ *computation* language has no I/O primitive inside!
    - you *choose* a *coordination* language by including (typically) `stdio.h` in C, and `iostream.h` or `stdio.h` in C++… *but other choices would be possible!*
    - whatever choice you make, the C/C++ *core language does not change*
  - a little farther: Linda & friends
    - Linda + C, Linda + Prolog, Linda + Java..

- **Syntactically, the coordination language "inherits" the "flavour" of the host language**
  - in Linda + C, rd/in/out are C functions, with typed (int/char*,...) arguments
  - in Linda + Prolog, they are predicates with term arguments
  - Linda + Java, they are suitable methods with suitable object arguments
  - …

Computation & Coordination Orthogonality: Benefits

- **Generality**
  - The same general-purpose coordination language can be used in different coordination contexts, *gluing together 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 programmer expertise in the same coordination context
 coordination models, languages and infrastructures

Coordination Mechanisms: some (not-so-good) examples

- **Message passing**, i.e. communication among peers
  - "message" is the only abstraction: no limitations, no constraints
  - no coordination abstraction, no intrinsic model of coordination: coordination just "occurs" thanks to suitable interaction protocols

- **Agent Communication Languages** enable communication
  - they *create the space* of inter-agent communication…
  - …but do not allow to constrain it
  - again, coordination is just "made of protocols"

- **Service-Oriented Architectures**
  - "service" is the only abstraction: very general.. but very simple
  - adopt the request/response pattern as the (only) pattern of coordination

- **Web servers**
  - based on the request/[representation]/response coordination pattern
  - multi-coordinated systems (objects, HTTP, applets, GUI, JavaScript..) often leading to "spaghetti coordination"
  - composition is more a source of chaos than an added value
Coordination Mechanisms: some (better) examples

- **Middleware** provide global properties across distributed systems
  - they fill the space of interaction with proper abstractions and nice shared features (interoperability, security, transactionality,...)
  - they can contain coordination abstractions... of any kind
    - since it can contain virtually anything, any middleware is specific

- **CORBA** has a more specific goal
  - to manage object interaction across a distributed system
  - many notions & conceptual tools (ORB, IDL...)
  - but, again, no specific model for coordination (apart from client/servant)
  - although any coordination abstraction/pattern can be supported

Focus is on enabling interaction (communication, interoperability) rather than constraining / ruling it.

Enabling vs Governing Interaction

- Enabling interaction is just the base, but is not enough

- Governing Interaction amounts at
  - ruling communication
  - providing concepts, abstractions, models, and mechanisms for the meaningful component integration
  - ruling and handling both component-component interaction and environment-component interaction

- Summing up, a coordination model should allow for:
  - ruling what components should/should not say /do at any time
  - possibly depending on what other components are saying/doing or have said/done (interaction history)
  - and/or on what is happening inside and outside the system
Coordination Models: Classification Criteria

- **Control-oriented vs. Data-oriented** models
  (Control-driven vs. Data-driven Models, [Papadopoulos and Arbab, 1998])
  - Control-oriented models focus on the acts of communication
  - Data-oriented models focus on the information exchanged during communication
  - Hybrid models also exist (see later...)

**Control-oriented**
- I/O ports, events & signals on state
- Explicit (poss. hierarchical) Coordinators:
  - create processes and communic. channels
  - determine (change) the topology of communication
- Point-to-point communication
- Coordination as configuration of topology
- Quite fine-grained
  - good for small-scale, closed systems
  - with a well-defined set of coordinables
  - needing fine-tuned control

**Data-oriented**
- Shared memory abstraction
- Processes emit & receive data/info
- Coordination by accessing, changing, synchronising on shared data
- Features:
  - expressive communication abstraction,
    leading to information-based design
  - space & time [name] uncoupling
- No more point-to-point communication
- Coordination as a first-class issue
  - good also for open systems
Control-oriented Models

- Coordinables typically open themselves to the external world and interact with it through *events* occurring on *well-defined input-output ports*.
- The *observable behaviour* of the coordinables from the coordination media's viewpoint is made of the *state changes and events* on such ports.
- The *coordination laws* establish how events and state changes can occur and how they should propagate (e.g., by means of *channels* linking ports).
- The coordination media focuses on the *topology* of the interaction space, *rather than the data exchanged* among the coordinables.
- Good when the focus of coordination is on the *management of the control flow*, rather than on the information exchanged.
  - TOOLBUS, Darwin, Regis, RAPIDE, ConCoord, MANIFOLD (the most known)

Data-oriented Models

- Coordinables interact *by exchanging data structures through the coordination media* which act as shared data (information) spaces.
  - The coordination laws establish how data should be represented and how they should be stored and extracted from the data space.
  - Coordination media do not perceive coordinables' state changes.
- Coordinables *synchronise, cooperate, and compete* based on the information available in the shared data space by *accessing, consuming and producing* information.
- Good when the focus of coordination is on the *information exchanged*, rather than the management of the control flow.
  - Law-governed Linda, Bauhaus Linda, IBM TSpaces, Sun JavaSpaces, GigaSpaces, ReSpecT & TuCSoN.
- Good for open systems, with an open (possibly unknown) set of coordinables.
  - with autonomous flows of control…
Hybrid Coordination Models

- **Control-oriented** and **data-oriented** models have both pros and cons

- **Hybrid coordination models** aim to get "the best of both worlds"
  - typically enriching a model of one family by integrating some elements from the other
  - For instance, the ReSpecT model (& language) that we will see later:
    - adopts shared interaction spaces (a typical data-oriented choice)
    - with Linda-like coordination primitives (again, inherently data-oriented),
    - but expresses its coordination laws in a control-oriented fashion, which is an essential feature to guarantee the desired media properties.

The Typical Cases

- **Control-oriented** models and languages
  - typical model: IWIM
  - typical language: Manifold

- **Data-oriented** models and languages
  - typical model: Tuple Space
  - typical language: Linda (& the like)

- **Hybrid** models and languages
  - typical model: Tuple Centre
  - typical language: ReSpecT (used in the TuCSon infrastructure)
Perhaps.. history repeats?

- **Sequential programming**
  - first: imperative spaghetti-style, with "goto"s
  - then, control came: structured programming (procedure-oriented)
  - after, data came: OOP (data-oriented)

- **Coordination programming**
  - first: "coordination..what?" (procedure call)
  - then, control came: control-oriented coordination
  - after, data came: data-oriented coordination

The Tuple Space meta-model

- **Coordinables** synchronise, cooperate, and compete based on *tuples*
  - ordered collections of (possibly heterogeneous) information chunks available in the *tuple space*

- **Coordination medium**: the *tuple space*

- **Coordinables** operate by **associatively** access, consume and produce tuples in the tuple space

From the language viewpoint:
- the *communication language* adopts a tuple-based syntax (whose types and details depend on the host language)
- the *coordination language* is defined by the set of operations for accessing, retrieving and inserting tuples
The Linda case (Gelernter, 1985) (1)

- **Metamodel**: the *communication language* adopts a tuple-based syntax (whose types and details depend on the host language)

- **Linda communication language**:
  - *tuples* as ordered collections of possibly heterogeneous data chunks
    - Linda+Prolog: `p(1), printer('HP',dpi(300)), [0,0.5]`
    - Linda+C: `("donald", 1966),`
  - *tuple templates* (also called *anti-tuples*) to specify set of [desired] tuples
    - Linda+Prolog: `p(X), [N,P],`
    - Linda+C: `{name:string, i:integer}, (s:string, 1966)`
  - *tuple matching mechanism* to access/select tuples vs. templates
    - Linda+Prolog: unification
    - Linda+C: pattern matching

The Linda case (Gelernter, 1985) (2)

- **Metamodel**: the *coordination language* is defined by the set of operations for *accessing, retrieving* and *inserting* tuples
  (whose form, types and details depend on the host language)

- **Linda coordination language**:
  - `out(T)` inserts tuple `T` into the tuple space
    - Linda+Prolog: `out(p(1)), out(course(infoT2,teacher('Denti')))`
    - Linda+C: `out("Enrico Denti", 44)`
  - `in(TT)` retrieves from the tuple space a tuple matching the template `TT`
    - Linda+Prolog: `in(p(1)), in(p(Y)), in(course(infoT2,teacher(X)))`
    - Linda+C: `in(s:string, age:int)`
  - `rd(TT)` accesses from the tuple space a tuple matching the template `TT`
    - Linda+Prolog: `rd(p(1)), rd(p(Y)), rd(course(infoT2,teacher(X)))`
    - Linda+C: `rd(s:string, age:int)`
The Linda case (Gelernter, 1985) (2)

- **Metamodel**: the coordination language is defined by the set of operations for accessing, retrieving and inserting tuples (whose form, types and details depend on the host language).

- **Linda coordination language**:
  - `out(T)` inserts tuple T into the tuple space
  - `in(TT)` retrieves from the tuple space a tuple matching the template TT
  - `rd(TT)` accesses from the tuple space a tuple matching the template TT

  **Retrieving by REMOVING** (destructive retrieving)
  - Linda+Prolog: `in(p(1))`, `in(p(Y))`, `in(course(infoT2,teacher(X)))`
  - Linda+C: `in(s:string, age:int)`

  **Accessing by READING** (non-destructive access)
  - Linda+Prolog: `rd(p(1))`, `rd(p(Y))`, `rd(course(infoT2,teacher(X)))`
  - Linda+C: `rd(s:string, age:int)`

  **Both**:
  - **Suspensive semantics**: if no matching tuples exist in the tuple space, operation execution is suspended until a matching tuple is eventually found.
  - **Non-determinism**: if multiple tuples match the given template, the selected one is chosen non-deterministically.

The Linda case: an example

```
out(p(1))

p(1) -> in(p(X))

array(1,13.3) -> rd(array(_,13.3))

printer('HP',dpi(300)) -> out(printer('HP', dpi(300)))
```
Linda: coordination language extensions

- **Problem 1:** the *suspensive semantics* is not always what you want
  - sometimes, you might prefer just to be informed that such a tuple is not present, instead of remain blocked indefinitely

  That’s why the Linda *coordination language is extended* with the *predicative versions* of *in* and *rd*
  - two further operations are added, *inp* and *rdp*, which feature a *success/failure semantics* instead of the suspensive one
  - if no matching tuple is found, *they report a failure* without suspending

- **Problem 2:** the above operations always deal with *one* tuple at a time
  - some coordination problems call for handling multiple tuples altogether

  The Linda *coordination language is then further extended* in some *versions* with *bulk primitives* such as *in_all* and *rd_all*
  - *success semantics:* if no matching tuple exists, an empty set is returned
  - other bulk primitives have been proposed for special classes of problems

Linda model [& coordination language] extensions: Multiple Tuple Spaces

- A *single tuple space can become a bottleneck* for coordination

  So, extensions have been proposed that *extend the Linda coordination model* towards a *multiplicity of tuple spaces*
  - basic idea: each tuple space encapsulates *only a portion* of the overall coordination burden
  - *further step: distribution* – such tuple spaces could either be hosted on a single machine, or distributed across a network

  In turn, this requires an *extended coordination language* to express
  - the concept of *tuple space name per se*
  - in the case of distribution, also the *tuple space location* in the network
  - the *target tuple space* of an operation – that is, on which tuple space a primitive is supposed to operate

- Possible example: `ts@node ? out(p)`
  - *tuple space name*
  - *tuple space location*
  - *coordination primitive*
Tuple-based Coordination: Features

- Full separation between computation and coordination
- **Tuples**
  - tuples have a record-like (possibly typed) structure, but *no need of field names: they just aggregate knowledge* regarding some given item
  - tuples are just characterised by their *arity*, as well as the type and position of their arguments – and, of course, their information content
- **Associative access**
  - tuples in the tuple space are accessed *through their content & structure*, not by their name, address, or location
- **Generative communication**
  - until explicitly withdrawn, the tuples generated by coordinables *have an independent existence* in the tuple space – the lifetime of a tuple is unrelated to that of the coodinable that created it (like a post-it..)
  - a tuple is equally accessible to all the coordinables, but is bound to none

Generative Communication: consequences

- **Generative communication** implies **Communication orthogonality**
  - senders and receivers can interact *without having any prior knowledge* about each others

As a result, three key properties hold:

- **Space uncoupling:**
  - agents do not need to coexist *in the same place* in order to interact
- **Time uncoupling:**
  - agents do not need to coexist *in the same time* in order to interact
- **Name uncoupling:**
  - agents do not need to *know each other* in order to interact

**Thanks to these properties:**
- Classic communication & synchronisation patterns can be easily reproduced, usually with less effort
- Complex coordination patterns can be built based on 3 simple primitives
- The design approach scales with complexity
A simple example: sharing a pool of resources (1)

- **Problem**: some agents have to share a pool of common resources

- **The model**
  - each resource is represented by its `resNumber`
  - an available resource is expressed by a tuple such as `resource(resNumber)`

- **The protocol**
  - to access a resource in the pool, an agent asks for an `resource(N)` tuple
  - when a resource is available, it is assigned to the agent by removing the corresponding tuple – otherwise, the agent waits
  - conversely, to release the resource, the agent just re-inserts the above tuple back in the tuple space

- **Tuple Space initial state**
  - there are as many `resource(resNumber)` tuples in the space as resources in the pool
    - in the figure, six resources are shown

---

A simple example: sharing a pool of resources (2)

- **The protocol**
  - to access a resource in the pool, an agent asks for an `resource(N)` tuple
  - when a resource is available, it is assigned to the agent by removing the corresponding tuple – otherwise, the agent waits
  - conversely, to release the resource, the agent just re-inserts the above tuple back in the tuple space

- **The Agent code (in prolog)**

  ```prolog
  go :-
    in(resource(N)),
    doWhatYouWant(gotResource(N)),
    out(resource(N)), !, go. // cyclic agent
  ```

- **So, what?**
  - **good**: the agent protocol focuses on performing its task, not with choosing, competing, etc
  - **good**: coordination and synchronisation are mostly delegated to the coordination medium
A more classic example: Dining Philosophers (1)

- **Problem:**
  - at any time, each philosopher either thinks or eats spaghetti from a spaghetti bowl
  - in order to eat, a philosopher needs the pair of chopsticks at his left and his right
  - however, chopsticks are shared among adjacent philosophers – there are exactly \( N \) chopsticks for \( N \) philosophers
  - when a philosopher finishes eating and wants to restart thinking, it just gives his pair of chopsticks back to the table.

- **Issues:**
  - shared resources – two adjacent philosophers cannot eat simultaneously
  - starvation – if a philosopher eats all the time, the two adjacent will starve
  - deadlock – if every philosopher picks up the same (left) chopstick at the same time, all will wait indefinitely for the other (right) chopstick
  - fairness - if a philosopher releases a chopstick before the other one, it favours one of his adjacent philosophers over the other one.

A more classic example: Dining Philosophers (2)

- **The model**
  - the table is represented by the tuple space
  - chopsticks are represented as \( \text{chop}(i) \) tuples
    (philosopher \( k \) uses chopsticks \#(left) and \#(right))

- **The protocol**
  - in order to eat, philosopher \( k \) needs to get the chopstick pair \((k, k+1)\)mod\(N\), that is, the pair of tuples \( \text{chop}(i) \) and \( \text{chop}(i+1) \)
  - conversely, when over, it must re-inserts the above tuple back in the tuple space

- **Tuple Space initial state**
  - there are as many \( \text{chop}(i) \) tuples as philos (5)

- **The Agent code (in prolog)**

  ```prolog
  philo(I,J) :-
  think, 
in(chop(I)), in(chop(J)),
  eat_as_much_as_you_want,
  out(chop(I)), out(chop(J)),
  !, philo(I,J). \% cyclic agent
  ```

- **So, what?**
  - shared resources handles correctly, but..
  - …starvation, deadlock, fairness NOT AT ALL!
A more classic example: Dining Philosophers (3)

- The new goal
  - to prevent starvation and deadlock
  - to guarantee fairness
  - to solve: complicating the protocol

- Possible new protocols
  - step 1: giving back the first chopstick if the second is not free \(\rightarrow\) avoids deadlock...
  - but starvation and unfairness still possible
  - step 2: introducing a token to access the table, so that getting/releasing the chopstick pair becomes an atomic action
  - etc etc...

- The key points
  - the more you complicate the protocol, the bigger the coordination burden becomes
  - the coordination burden is put on top of agents rather than in the coordination medium, where it would conceptually belong.
  - agents spend more time (and code..) in managing coordination than their real tasks!
  - WHY?

A more classic example: Dining Philosophers (4)

Tuple space vs. tuple centre: pseudo-code for a C philosopher

(a) Tuple space

```c
while (true) {
  /* main cycle */
  Philo();
  /* acquisition */
  int(chop(1));
  int(chop(2));
  /* release */
  out(chop(1));
  out(chop(2));
}
```

(b) Tuple space (deadlock-free)

```c
while (true) {
  /* main cycle */
  Philo();
  /* acquisition */
  if (CanEat = false) {
    int(chop(1));
    int(chop(2));
    out(chop(1));
    out(chop(2));
  } else if (CanEat = true) {
    eat();
    /* release */
    out(chop(1));
    out(chop(2));
  }
}
```

Most of time & code used for managing coordination rather than their actual tasks
Dining Philosophers in Linda: the very reasons of difficulties

- **Coordination in Linda is limited to writing, reading, consuming, suspending on one tuple at a time**
  - If this behaviour of the coordination medium is good for your problem, ok
  - otherwise, the solution is unsatisfactory

- **New bulk primitives are not a general-purpose solution**
  - adding ad hoc primitives does not solve the problem in general
  - but above all, does not fit open scenarios
    - *should we add and add more and more primitives... forever??*
  - open scenarios have the opposite requirement
    - little, well-known, simple primitives for doing all

- **Coordination in Linda is typically performed via complex, ad-hoc protocols, which charge the burden of coordination upon coordinables**
  - against any software engineering principle (encapsulation, locality)
  - but above all, **against common sense!**
    - an agent is there to perform its task, not someone else’s!

Summing up: Tuple Space Limits

- **No distinction** between information actually conveyed by tuples and its representation
  - consequently, no way to separate how information is represented from how it is perceived by agents

- **Fixed behaviour** of the coordination medium: you either like it, or not!
  - Non-trivial coordination policies cannot be encapsulated in the coordination medium, so they turn up to be put in the wrong place (i.e. on the coordinables)
  - The coordination laws (mechanisms) are not powerful (general purpose) enough to express generic coordination policies

- **So, things go the wrong way:**
  - coordination policies realised as specialised agent interaction protocols
  - coordinables need to be ‘coordination-aware’
    - coordinables cannot abstract from the coordination details (**subjective coordination**)
  - violation of encapsulation of interaction management/coordination
  - coordination scalability problems (complexity, heterogeneity...)

- **We definitely deserve a better world ☺☺ ☺☺... but how?**
A novel approach: Encapsulating the Glue

- **The behaviour of the coordination media should not be fixed once and for all**
  - the "either get this behaviour, or do it by yourself" aut-aut is not acceptable

- **The behaviour of the coordination media should actually be programmable**
  - it should be able to capture the issues posed by (any) coordination problem
    - need for Turing-equivalence of media’s computational model

- **Programming the coordination media.. how ?**
  - perhaps adding new ad-hoc primitives..?
    - not general, not reusable, causes pollution of the primitive space
  - perhaps changing the semantics of the old ones..?
    - not advisable – people need certainty when thinking & designing..

- **Coordination media as the natural places to encapsulate the glue**
  - the coordination burden should stay where it conceptually belongs
  - the right/natural places where to encapsulate solutions to coordination problems:
    - solutions represented as coordination policies, and enacted by the media behaviour

- **What do we need?**
  - a general (Turing-equivalent) computational model for the coordination media
  - an expressive programming language to specify the media behaviour

Encapsulating the Glue: from Tuple Spaces to Tuple Centres

- **The new requirements**
  - no new ad-hoc primitives
  - no changes to the syntax and semantics of the standard Linda primitives
  - no changes even to the matching mechanism (in a given context)
  - support for multiple tuple spaces, also accounting for tuple space distribution
  - support for programming the coordination media, to embody coordination policies
    - to this end: ability to associate coordinative behaviours to coordination events

- **Tuple Centres ::= Programmable Tuple Spaces**
  - the behaviour of the coordination media 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 are no longer fixed, but tailored to the specific coordination needs

- **The above requirements are all met:**
  - tuple centres maintain the same tuple space interface (same primitive set, the basic Linda primitives behaving exactly as usual)
  - so, the coordinables perceive tuple centres just as standard tuple spaces
  - …but their behaviour can be enriched so as to encapsulate the coordination rules.
A **tuple centre** is a tuple space enhanced with a **behaviour specification**, which defines its behaviour in response to **interaction events**.

The **behaviour specification** of tuple centre:
- is expressed in terms of a **reaction specification language**
- associates any **tuple centre event** to a (possibly empty) set of computational activities, called **reactions**
  - if such a set is empty, the tuple centres behaves like a tuple space
  - otherwise, a tuple centre can behave very differently w.r.t. a tuple space, according to the coordination laws required by the problem to be solved.

From the language viewpoints:
- **nothing changes** for the coordinables, which still see & use the standard Linda primitives in the usual manner
- **inside** the coordination media, instead, a **reaction specification language**:
  - enables the definitions of **computational activities (reactions)** in the medium
  - provides a way to **associate reactions to the events** occurring in the medium

---

**Tuple Centres (Omicini & Denti, 2001) (2)**

- **Each reaction** must be enabled to:
  - access and modify the current tuple centre state
    - adding, reading, removing tuples – or even checking for their absence
  - access any information related to the triggering event, which is made fully observable
    - the performing agent, the primitive invoked, the tuple involved, etc
  - invoke link primitives upon other tuple centres (linkability property)

- **The tuple centre computational model** must guarantee:
  - Turing-equivalence, so as that any coordination law can be expressed
  - that all the “relevant events” are fully perceived and considered
  - that each reaction has a transactional semantics – **either succeed, or yield no effect**
  - that the behaviour observed by coordinables from the outside after an operation **is still perceived as a single transition of state**, like in a standard tuple space

- **Within this framework**, in principle:
  - a multiplicity of reaction specification languages could be envisioned
  - different tuple centre computational models could be imagined
Tuple Centres as a form of Hybrid Coordination

- **Hybrid coordination models** aim to "get the best" of both data-oriented and control-oriented models.

- **Tuple centres are a case of hybrid coordination model**, since:
  - they are basically an information-driven coordination medium
    - and are perceived as such by coordinables
  - but at the same time, they provide typical control-driven features, like:
    - the full observability of events
    - the ability to selectively react to events
    - the ability to implement coordination rules
      - by manipulating the interaction space

---

Coordination Models, Languages and Infrastructures

Enrico Denti
enrico.denti@unibo.it

ALMA MATER STUDIORUM – Università di Bologna

Scuola di Dottorato in Ingegneria e Scienza dell’Informazione
6 luglio 2010
ReSpecT Tuple Centres

ReSpecT tuple centres are an instance of tuple centres, where:
- tuples are logic tuples
- the reaction specification language, ReSpecT, is logic-based
- from the viewpoint of the coordinables, any operation follows a two-phase protocol
  - first, the operation is invoked on the tuple centre
  - then, the operation is served internally in the tuple centre
  - finally, a "result" is returned by the tuple centre back to the performing coordinable.

The ReSpecT computational model
- the ReSpecT virtual machine operates according to the following main cycle:
  - when a primitive invocation reaches the tuple centre, all the related reactions (if any) are triggered, and then executed in a non-deterministic order
    - reactions can trigger further reactions, in a chain
  - once all such reactions have been executed, the primitive is served in the same way as in standard Linda
  - upon completion of the primitive, the further corresponding reactions (if any) are triggered, and then executed in a non-deterministic order (*)
    - reactions can trigger further reactions, in a chain
  - once all the reactions have been executed, the main cycle restarts.

If no reactions are specified, the tuple centre behaves like a standard tuple space.
ReSpecT Tuple Centres: State transitions & Perception

The ReSpecT computational model:
- has been proven to be Turing-equivalent, as required by any tuple centre model
- is structured in such a way that, from the agent’s viewpoint, the invocation of a
tuple centre primitive is perceived as the sum of the effects of the primitive itself
and of all the triggered reactions (each behaving as a single atomic transaction)
  - the "triggered reactions" include those triggered indirectly,
as a consequence of other, previous reactions, in a chain
- but, as required, such effects are perceived altogether as a single-step transition
of the tuple centre state, as in any standard tuple space

Tuple centres vs. tuple spaces: state transitions
- in standard tuple spaces, state transitions are constrained to adding, reading or
deleting one single tuple – and are unmodifiable
- in tuple centres, state transitions be made as complex as needed – and are
programmable by the designer

Tuple centres vs. tuple spaces: perception & decoupling
- in standard tuple spaces, the agent’s view of the tuple space is fixed
- in tuple centres, the agent’s view of the tuple centre is decoupled from the actual
state of the tuple centre – the two are related according to the coordination laws

ReSpecT: from the origins to.. the future


- ReSpecT 1st version (1.4)
  - first version of the model & language (Omicini & Dentì, 2001)
  - embedded in the TuCSoN infrastructure, not available alone
  - last stable version: ReSpecT 1.4 (Ricci, 2004) based on tuProlog 1.2.1

- ReSpecT 2nd version (1.5)
  - model and language untouched
  - refactored version, separated from TuCSoN, available stand-alone
  - last stable version: ReSpecT 1.5 (Ricci & Casadei, 2006) based on tuProlog 1.4

- ReSpecT 3rd version
  - model & language extended with timed reactions (Omicini & Ricci & Viroli, 2005)
  - model & syntax revised in the A&A perspective (Omicini, 2007)
  - only prototype implementations

- ReSpecT 4th version (2.2)
  - model & language revised with situatedness (Casadei & Omicini, 2008-2009)
  - current version: ReSpecT 2.2 (Casadei, 2008) - console available also via Java Web Start

- ReSpecT 5th version (2.3, expected "soon")
  - model and language untouched
  - based on tuProlog 2.2 (revised engine, exceptions, multi-threading..)
ReSpecT Tuple Centres: Example (1/4)

**Requirements:**
- Whenever a tuple $T$ is inserted in the space, a tuple $\text{backup}(T)$ should be also produced.
- Whenever a tuple $p(X)$ is inserted, the tuple $\text{total_tuples}(N)$ should be properly updated.

**Tuple space initial state:**
- At the beginning, just one $p/1$ tuple is present, so the counter tuple is set to $1$: $\text{total_tuples}(0)$

ReSpecT 1.5:

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))}).

ReSpecT 2.2:

reaction(out(T), {operation, invocation}, {out(backup(T))}).

reaction(out(p(X)), {operation, invocation}, {in(total_tuples(N)), N1 is N + 1, out(total_tuples(N1))}).

Enrico Denti (Università di Bologna)  Coordination Models, Languages and Infrastructures  Bologna, 2/7/2010 55

Enrico Denti (Università di Bologna)  Coordination Models, Languages and Infrastructures  Bologna, 2/7/2010 56

Enrico Denti (Università di Bologna)  Coordination Models, Languages and Infrastructures  Bologna, 2/7/2010 57

ReSpecT Tuple Centres: Example (2/4)
reaction( out(T), (operation, invocation), { out( backup(T) ) }).

reaction( out(p(X)), (operation, invocation), { in(total_tuples(N)), N1 is N + 1, out(total_tuples(N1)) }).

The coordinable inserts the tuple
hungry('Enrico')
in tuple centre. Trying to read it back, the tuple is actually there...

...but also backup(hungry('Enrico')) is there! The tuple centre reacted to the original out, applying the desired coordination policy.

reaction( out(T), (operation, invocation), { out( backup(T) ) }).

reaction( out(p(X)), (operation, invocation), { in(total_tuples(N)), N1 is N + 1, out(total_tuples(N1)) }).

Let us check total_tuples(N): as agreed, it is initially 1.

Now, let us insert a p/1 tuple such as p(3)
The operation succeeds, and if we now check total_tuples(N) again, we find it is 2 !

It is also worth noting that, since out(p(3)) matches both reactions, we also find the backup(p(3)) in the space (but just one...)!
ReSpecT: the Language

- A ReSpecT program is a set of ReSpecT reactions
  - triggered and executed according to the ReSpecT computational model

- A ReSpecT reaction
  - is expressed as a logic tuple
    - in ReSpecT 1.x, of the form `reaction(Event, Body)`
    - in ReSpecT 2.2, of the form `reaction(Event, [Guard], Body)`
  - which associates the event `Event` to the reaction body `Body`
    - in ReSpecT 2.2, a much broader set of `Events` can be specified, conditioning the reaction execution to the set of (optional) ` Guards`
    - ReSpecT 1.x allowed for some simple guards inside the reaction `Body`

- In turn, `Body` is a sequence of reaction goals, which:
  - are executed sequentially, with a transactional semantics (all or nothing)
  - can manipulate tuples in this tuple centre and in other tuple centres (linkability)
  - can perform simple Prolog-like arithmetic & logic operations
  - access event properties – in ReSpecT 2.2, also environment & time properties

---

ReSpecT 1.x Syntax: simplicity.. with some weaknesses

Main ReSpecT predicates for reactions

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>out(T)</code></td>
<td>succeeds and inserts tuple T into the tuple centre</td>
</tr>
<tr>
<td><code>in(T)</code></td>
<td>succeeds, if a tuple T matching template TT is found in the tuple centre</td>
</tr>
<tr>
<td><code>in(T, TT)</code></td>
<td>succeeds, if a tuple T matching template TT is found in the tuple centre</td>
</tr>
<tr>
<td><code>eq(T, TT)</code></td>
<td>succeeds, if T is equal to TT</td>
</tr>
<tr>
<td><code>eq(T, TT)</code></td>
<td>succeeds, if T matches the tuple involved by the current communication event</td>
</tr>
<tr>
<td><code>current_agent(A)</code></td>
<td>succeeds, if A matches the identifier of the agent which triggered the current communication event</td>
</tr>
<tr>
<td><code>current_agent(A)</code></td>
<td>succeeds, if A matches the identifier of the agent which triggered the current communication event</td>
</tr>
<tr>
<td><code>current_agent(A)</code></td>
<td>succeeds, if A matches the identifier of the agent which triggered the current communication event</td>
</tr>
<tr>
<td><code>current_agent(A)</code></td>
<td>succeeds, if A matches the identifier of the agent which triggered the current communication event</td>
</tr>
<tr>
<td><code>current_agent(A)</code></td>
<td>succeeds, if A matches the identifier of the agent which triggered the current communication event</td>
</tr>
<tr>
<td><code>current_agent(A)</code></td>
<td>succeeds, if A matches the identifier of the agent which triggered the current communication event</td>
</tr>
<tr>
<td><code>current_agent(A)</code></td>
<td>succeeds, if A matches the identifier of the agent which triggered the current communication event</td>
</tr>
<tr>
<td><code>current_agent(A)</code></td>
<td>succeeds, if A matches the identifier of the agent which triggered the current communication event</td>
</tr>
<tr>
<td><code>current_agent(A)</code></td>
<td>succeeds, if A matches the identifier of the agent which triggered the current communication event</td>
</tr>
<tr>
<td><code>current_agent(A)</code></td>
<td>succeeds, if A matches the identifier of the agent which triggered the current communication event</td>
</tr>
<tr>
<td><code>current_agent(A)</code></td>
<td>succeeds, if A matches the identifier of the agent which triggered the current communication event</td>
</tr>
<tr>
<td><code>current_agent(A)</code></td>
<td>succeeds, if A matches the identifier of the agent which triggered the current communication event</td>
</tr>
<tr>
<td><code>current_agent(A)</code></td>
<td>succeeds, if A matches the identifier of the agent which triggered the current communication event</td>
</tr>
<tr>
<td><code>current_agent(A)</code></td>
<td>succeeds, if A matches the identifier of the agent which triggered the current communication event</td>
</tr>
<tr>
<td><code>current_agent(A)</code></td>
<td>succeeds, if A matches the identifier of the agent which triggered the current communication event</td>
</tr>
</tbody>
</table>

Event properties limited to tuple, agent, operation and tc name

- `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

---

Enrico Denti (Università di Bologna)  Coordination Models, Languages and Infrastructures  Bologna, 2/7/2010
ReSpect 2.x Syntax:
full event observability and situatedness

- Explicit guard predicates
- Time and environment events also considered
- Environment properties fully accessible
- Uniform names btw internal & external operations
- Full event observability
- prime cause vs direct cause of an event

ReSpect 1.5 vs 2.2: Language differences (1)

- **Operation names**
  - In ReSpect 2.x, primitive operation names are uniform both inside and outside reactions
    - although \texttt{rd/rdp} and \texttt{in/inp} obviously collapse to the same non-suspendive semantics
  - In ReSpect 1.x, primitive operation names inside reactions had a "\texttt{r}" suffix
    - Actually, \texttt{cd_r} and \texttt{in_r} replaced the \texttt{rd/rdp} and \texttt{in/inp} pairs, respectively
  - In ReSpect 1.x, the \texttt{no_r} reaction primitive had no agent counterpart

- **Operation phases**
  - In ReSpect 2.x, all primitives follow a two-phase, request/response protocol
  - In ReSpect 1.x, the \texttt{out} primitive was an exception
    - It had the request phase only – no response, no ack
    - Beware! possible subtle differences in reaction behaviour when porting old programs!

- **Operation guards**
  - In ReSpect 2.x, a large set of guards can be specified (fine-grained control)
    - Not only operation phases and result, but also origin, target, interna, external...
  - In ReSpect 1.x, guards were limited to four basic cases
    - The request / response phase of the operation (\texttt{pre/post})
    - The success / failure of the operation (\texttt{success/failure})
### ReSpect 1.5 vs 2.2: Language differences (1)

#### Operation names
- In ReSpect 2.x, primitive operation names are uniform both inside and outside reactions although `rd/rdp` and `in/inp` obviously collapse to the same non-suspensive semantics
- In ReSpect 1.x, primitive operation names inside reactions had a "_r" suffix although `rd_r` and `in_r` replaced the `rd/rdp` and `in/inp` pairs, respectively
- In ReSpect 1.x, the `no_r` reaction primitive had no agent counterpart

#### Operation phases
- In ReSpect 2.x, all primitives follow a two-phase, request/response protocol
- In ReSpect 1.x, the `out` primitive was an exception – it had the request phase only – no response, no ack

#### Operation guards
- In ReSpect 2.x, a large set of guards can be specified (fine-grained control) not only operation phases and result, but also origin, target, context, etc.
- In ReSpect 1.x, guards were limited to four basic cases the request/response phase of the operation (`pre`/`post`) the success/failure of the operation (`success`/`failure`) the `out` operation only has the request phase.

#### Linkability
(i.e. the capability of triggering an event on another tuple centre) In ReSpect 2.x, linkability is fully supported (any operation is usable for linking) In ReSpect 1.x, linkability was limited to `out` operations (`out_tc`)

#### Event information
- In ReSpect 2.x, any event-related information is available (fine-grain control) on any kind of operation, on any kind of phase, etc including `origin` and `destination` of any operation (`from_tc`, `from_agent`, etc) separating the `prime` cause of an event from its `direct` cause including `time`
- In ReSpect 1.x, event-related information was limited to four basic cases `current_agent`, `current_op`, `current_tuple`, `current_tc`

#### Situatedness
- In ReSpect 2.x, also time and environment properties can be sensed & handled
  - `timed` reaction
  - `sensing/acting` on environment properties (`get`/`set` – see after `...`)
- In ReSpect 1.x, no such concept existed.

---

### ReSpect 1.5 vs 2.2: Language differences (2)

#### Linkability
(i.e. the capability of triggering an event on another tuple centre)

<table>
<thead>
<tr>
<th>ReSpect 1.x</th>
<th>ReSpect 2.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linkability was limited to <code>out</code> operations (<code>out_tc</code>)</td>
<td>Linkability is fully supported (any operation is usable for linking)</td>
</tr>
</tbody>
</table>

#### Event information
- In ReSpect 2.x, any event-related information is available (fine-grain control) on any kind of operation, on any kind of phase, etc including `origin` and `destination` of any operation (`from_tc`, `from_agent`, etc) separating the `prime` cause of an event from its `direct` cause including `time`
- In ReSpect 1.x, event-related information was limited to four basic cases `current_agent`, `current_op`, `current_tuple`, `current_tc`

#### Situatedness
- In ReSpect 2.x, also time and environment properties can be senses & handled
  - `timed` reaction
  - `sensing/acting` on environment properties (`get`/`set` – see after `...`)
- In ReSpect 1.x, no such concept existed.
ReSpecT 1.x vs 2.x Syntax:
a quick "how-to" for program conversion

<table>
<thead>
<tr>
<th>issue</th>
<th>example</th>
<th>conversion to ReSpecT 2.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>general reaction syntax</td>
<td>reaction(op, body)</td>
<td>Change reaction/2 to reaction/3 adding guards; change body accordingly</td>
</tr>
<tr>
<td>reactions to communication operations</td>
<td>reaction(in(p), ...)</td>
<td>Operation name unchanged, but add the &quot;operation&quot; guard to the guard section</td>
</tr>
<tr>
<td>reactions to internal TC operations</td>
<td>reaction(in_r(p), ...)</td>
<td>Remove &quot;r&quot; from operation name, but be sure to add the &quot;internal&quot; guard to the guard section</td>
</tr>
<tr>
<td>reaction guards</td>
<td>reaction(op, {pre, goals})</td>
<td>Remove such guards from body, and put them into the new guard section Consider using the new guard names instead of pre/post (available as aliases)</td>
</tr>
<tr>
<td>powers of reactions to out operations</td>
<td>reaction(out(p), body)</td>
<td>• in ReSpecT 2.x, out are two-phase operations, have both phases • the emitted tuple is available in the post phase only – not before To prevent reactions to be executed twice, always add the proper guard (typically, &quot;completion&quot;) to the guard section</td>
</tr>
</tbody>
</table>

ReSpecT 1.5:
reaction( out(T), { out_r( backup(T)) }).
reaction( out(goofy), { in_r( goofy ) } ).

ReSpecT 2.2:
reaction( out(T), (operation, invocation), { out( backup(T)) }).
reaction( out(goofy), (operation, completion), { in_r( goofy ) } ).

Dining Philosophers... again (1)

- Decoupling agent perception & resource representation
  - with tuple centres, the agent’s perception can be decoupled from the actual TC state: the two are related by the coordination laws

- Resource representation
  - chopsticks are single resources, so their natural representation is still as chop(i) tuples

- Agent’s protocol
  - philosophers handle pairs of chopsticks, so their natural approach is asking for/releasing chopstick pairs, e.g. chops(1, 2) tuples

- Highlights
  - the coordination burden goes where it conceptually belongs – the coordination medium
  - agents focus on their tasks (eat & think), not in managing coordination
  - coordination laws bridge between agents’ view and the actual resource representation, guaranteeing the required properties (no starvation, no deadlock, fairness, etc.)
Dining Philosophers... again (2)

Tuple space vs. tuple centre: pseudo-code for a C philosopher

<table>
<thead>
<tr>
<th>(a) Tuple space</th>
<th>(b) Tuple space (deadlock-free)</th>
<th>(c) Tuple centre</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>while (true) {</code></td>
<td><code>while (true) {</code></td>
<td><code>while (true) {</code></td>
</tr>
<tr>
<td><code>sleep();</code></td>
<td><code>sleep();</code></td>
<td><code>sleep();</code></td>
</tr>
<tr>
<td><code>/* acquisition */</code></td>
<td><code>/* acquisition */</code></td>
<td><code>/* acquisition */</code></td>
</tr>
<tr>
<td><code>in(chop(C1));</code></td>
<td><code>in(chop(C1));</code></td>
<td><code>in(chop(C1));</code></td>
</tr>
<tr>
<td><code>/* release */</code></td>
<td><code>/* release */</code></td>
<td><code>/* release */</code></td>
</tr>
<tr>
<td><code>out();</code></td>
<td><code>out();</code></td>
<td><code>out();</code></td>
</tr>
</tbody>
</table>
| `}` | `}` | `}`

Dining Philosophers... again (3)

ReSpecT specification for the coordination of the Dining Philosophers

```
reaction(in(chops(C1,C2)), ( (pre, out(required(C1,C2))) ));  // (1)
reaction(in(chops(C1,C2)), ( post, in(required(C1,C2))) ));  // (2)
reaction(out(r(chop(C1))), ( r, required(C1,C2) ));  // (3)
  in(chop(C1)), in(r(chop(C1))), out(r(chops(C1,C2))) ) ;
reaction(out(chops(C1,C2)), ( in.r(chops(C1,C2)), out.r(chop(C1,C2))) ));  // (4)
reaction(out(r(chop(C1))), ( r, required(C1,C2) ));  // (5)
  in(chop(C1)), in.r(chop(C1)), out.r(chops(C1,C2)) ) ;
reaction(out(r(chop(C2))), ( r, required(C1,C2) ));  // (6)
  in(chop(C1)), in.r(chop(C2)), out.r(chops(C1,C2)) ) ;
```

- Reaction (1), triggered in the invocation phase (pre guard) of the `in(chops(C1,C2))` operation, inserts a `required(C1,C2)` tuple in the space, to represent an incoming request for that chopstick pair.

- Reaction (3) reacts to the insertion of the `required(C1,C2)` tuple made by reaction (1) (reaction chain). If the required chopsticks (C1) and (C2) are both present, reaction (3) consumes them atomically, and produces the required `chops(C1,C2)` tuple expected by the corresponding philosopher, which is then served. If one of the two chops is missing, the reaction fails as a whole, yielding no result at all, as if it had never executed (transactional semantics): the philosopher will wait. (deadlock prevention)

- Reaction (2) reacts to the serving phase of the philosopher initial request (post guard), deleting the `required(C1,C2)` tuple from the space, thus representing that the request has been successfully served.

Coordination laws in ReSpecT 1.x

Most of time & code used for managing coordination rather than their actual tasks

Coordination reduced to the very minimum

Agent time & code on actual tasks

Tuple centre-based coordination for a C-like philosopher

Performance tests
Dining Philosophers... again (4)

ReSpecT specification for the coordination of the Dining Philosophers

\[
\begin{align*}
\text{reaction(in(chops(C1,C2)), \{ \text{pres}, \text{out}(\text{required}(C1,C2)) \}) }, & \quad (1) \\
\text{reaction(in(chops(C1,C2)), \{ \text{post}, \text{in}(\text{required}(C1,C2)) \}) }, & \quad (2) \\
\text{reaction(out.r(\text{required}(C1,C2)), \{ \text{in.r}(\text{chop}(C1)), \text{in.r}(\text{chop}(C2)), \text{out.r}(\text{chops}(C1,C2)) \}) }, & \quad (3) \\
\text{reaction(out.chops(C1,C2), \{ \text{in.r}(\text{chops}(C1,C2)), \text{out.r}(\text{chop}(C1)), \text{out.r}(\text{chop}(C2)) \}) }, & \quad (4) \\
\text{reaction(out.r.chop(C1), \{ \text{in.r}(\text{required}(C1,C)), \text{in.r}(\text{chop}(C1)), \text{out.r}(\text{chops}(C1,C)) \}) }, & \quad (5) \\
\text{reaction(out.r.chop(C2), \{ \text{in.r}(\text{required}(C,C2)), \text{in.r}(\text{chop}(C)), \text{out.r}(\text{chops}(C,C2)) \}) }, & \quad (6)
\end{align*}
\]

Coordination laws in ReSpecT 1.x

Reaction (4) handles the release of the chopstick pairs: when philosopher performs the `out(chops(C1,C2))` operation, it atomically consumes the `chops(C1,C2)` tuple (inserted by the philosopher's `out`) and re-inserts the single tuples `chop(C1)` and `chop(C2)` in the space.

Finally, reactions (5) and (6), both triggered by the insertion of tuple by reaction (4) (reaction chain), serve the pending requests which can be satisfied after a chopstick release. If a request exist for one of the chopstick become available (registered in a `required(C1,C)` or a `required(C,C2)` tuple), and the other needed chopstick is present, these reactions, like reaction (3) above, atomically replace the single `chop(i)` and `chop(j)` tuples by the `chops(i,j)` expected by the philosopher.

Dining Philosophers... again (5)

ReSpecT specification for the coordination of the Dining Philosophers

\[
\begin{align*}
\text{reaction(in(chops(C1,C2)), \{ \text{pres}, \text{out}(\text{required}(C1,C2)) \}) }, & \quad (1) \\
\text{reaction(in(chops(C1,C2)), \{ \text{post}, \text{in}(\text{required}(C1,C2)) \}) }, & \quad (2) \\
\text{reaction(out.r(\text{required}(C1,C2)), \{ \text{in.r}(\text{chop}(C1)), \text{in.r}(\text{chop}(C2)), \text{out.r}(\text{chops}(C1,C2)) \}) }, & \quad (3) \\
\text{reaction(out.chops(C1,C2), \{ \text{in.r}(\text{chops}(C1,C2)), \text{out.r}(\text{chop}(C1)), \text{out.r}(\text{chop}(C2)) \}) }, & \quad (4) \\
\text{reaction(out.r.chop(C1), \{ \text{in.r}(\text{required}(C1,C)), \text{in.r}(\text{chop}(C1)), \text{out.r}(\text{chops}(C1,C)) \}) }, & \quad (5) \\
\text{reaction(out.r.chop(C2), \{ \text{in.r}(\text{required}(C,C2)), \text{in.r}(\text{chop}(C)), \text{out.r}(\text{chops}(C,C2)) \}) }, & \quad (6)
\end{align*}
\]

Coordination laws in ReSpecT 1.x

Coordination laws in ReSpecT 2.x

Notice the new guards (in blue) and the uniformity of internal tuple centre operations (`in`, `out`, `rd` instead of `in_r`, `out_r`, `rd_r`).

\[
\begin{align*}
\text{reaction(in(chops(C1,C2)), \{ \text{invocation}, \text{operation} \}), & \quad (\text{out}(\text{chops}(C1,C2)))}, & \quad (1) \\
\text{reaction(in(chops(C1,C2)), \{ \text{completion}, \text{operation} \}), & \quad (\text{in}(\text{required}(C1,C2)))}, & \quad (2) \\
\text{reaction(out(\text{required}(C1,C2)), \{ \text{internal} \}), & \quad (\text{in}(\text{chop}(C1)), \text{in}(\text{chop}(C2)), \text{out}(\text{chops}(C1,C2)))}, & \quad (3) \\
\text{reaction(out.chops(C1,C2), \{ \text{internal} \}), & \quad (\text{in}(\text{chops}(C1,C2))), & \quad (4) \\
\text{reaction(out.r.chop(C1), \{ \text{internal} \}), & \quad (\text{in}(\text{chop}(C1)), \text{in}(\text{chop}(C2)), \text{out}(\text{chops}(C1,C))}), & \quad (5) \\
\text{reaction(out.r.chop(C2), \{ \text{internal} \}), & \quad (\text{in}(\text{chop}(C)), \text{in}(\text{chop}(C2)), \text{out}(\text{chops}(C,C2))))}, & \quad (6)
\end{align*}
\]
Dining Philosophers: step-by-step inside the TC (1)

Initial situation

reaction(in(chops(C1,C2)), (invocation, operation ), (out(required(C1,C2)))).
reaction(in(chops(C1,C2)), (completion, operation ), (in(required(C1,C2)))).
reaction(out(required(C1,C2)), (internal), (in(chop(C1)), in(chop(C2)), out(chops(C1,C2)))).
reaction(out(chops(C1,C2)), (completion, operation), (in(chops(C1,C2)), out(chop(C1)), out(chop(C2)))).
reaction(out(chop(C1)), (internal), (rd(required(C1,C))), in(chop(C1)), in(chop(C)), out(chops(C1,C)))).
reaction(out(chop(C2)), (internal), (rd(required(C,C2)), in(chop(C)), in(chop(C2)), out(chops(C,C2)))).

behaviour specification

Dining Philosophers: step-by-step inside the TC (2a)

Philo#2 wants to eat

reaction(in(chops(C1,C2)), (invocation, operation ), (out(required(C1,C2)))).
reaction(in(chops(C1,C2)), (completion, operation ), (in(required(C1,C2)))).
reaction(out(required(C1,C2)), (internal), (in(chop(C1)), in(chop(C2)), out(chops(C1,C2)))).
reaction(out(chops(C1,C2)), (completion, operation), (in(chops(C1,C2)), out(chop(C1)), out(chop(C2)))).
reaction(out(chop(C1)), (internal), (rd(required(C1,C)), in(chop(C1)), in(chop(C)), out(chops(C1,C)))).
reaction(out(chop(C2)), (internal), (rd(required(C,C2)), in(chop(C)), in(chop(C2)), out(chops(C,C2)))).
Dining Philosophers: step-by-step inside the TC (2b)

**Philo#2 wants to eat**

**behaviour specification**

- Philo#2 wants to eat
- Philo#2 got chopsticks!

```
reaction(in(chops(C1,C2)), (invocation, operation ), (out(required(C1,C2)))).
reaction(in(chops(C1,C2)), (completion, operation ), (in(required(C1,C2)))).
reaction(out(required(C1,C2)), (internal), (in(chop(C1)), in(chop(C2)), out(chops(C1,C2)))).
reaction(out(chops(C1,C2)), (completion, operation), (in(chops(C1,C2)), out(chop(C1)), out(chop(C2)))).
reaction(out(chop(C1)), (internal), (rd(required(C1,C)), in(chop(C1)), in(chop(C)), out(chops(C1,C)))).
reaction(out(chop(C2)), (internal), (rd(required(C,C2)), in(chop(C)), in(chop(C2)), out(chops(C,C2)))).
```

```
behaviour specification

Philo#2 got chopsticks!

```

Dining Philosophers: step-by-step inside the TC (2c)

**Philo#2 got chopsticks!**

**behaviour specification**

- Philo#2 wants to eat
- Philo#2 got chopsticks!

```
reaction(in(chops(C1,C2)), (invocation, operation ), (out(required(C1,C2)))).
reaction(in(chops(C1,C2)), (completion, operation ), (in(required(C1,C2)))).
reaction(out(required(C1,C2)), (internal), (rd(required(C1,C)), in(chop(C1)), in(chop(C)), out(chops(C1,C)))).
reaction(out(chops(C1,C2)), (completion, operation), (in(chops(C1,C2)), out(chop(C1)), out(chop(C2)))).
reaction(out(chop(C1)), (internal), (rd(required(C,C2)), in(chop(C1)), in(chop(C2)), out(chops(C,C2)))).
reaction(out(chop(C2)), (internal), (rd(required(C,C2)), in(chop(C1)), in(chop(C2)), out(chops(C,C2)))).
```
Dining Philosophers: step-by-step inside the TC (2d)

**Philo#2 eating...**

```
reaction(in(chops(C1,C2)), (invocation, operation ), (in( required(C1,C2)))).
reaction(in(chops(C1,C2)), (completion, operation ), (in( required(C1,C2)))).
reaction(out(required(C1,C2)), (internal ), (in(chop(C1)), in(chop(C2)), out(chops(C1,C2)))).
reaction(out(chops(C1,C2)), (completion, operation), (in(chops(C1,C2)), out(chop(C1)), out(chop(C2)))).
reaction(out(chop(C1)), (internal ), (rd(required(C1,C)), in(chop(C1)), in(chop(C)), out(chops(C1,C)))).
reaction(out(chop(C2)), (internal ), (rd(required(C,C2)), in(chop(C)), in(chop(C2)), out(chops(C,C2)))).
```

behaviour specification

Philo#2 eating...

```
.. and Philo#2 goes on eating as much as she wants (just for $10 !)

.. but eventually she’ll get tired of eating (and will start thinking of a diet..)
```

Dining Philosophers: step-by-step inside the TC (3)
Dining Philosophers: step-by-step inside the TC (4a)

Philo#2 gives chops back

**t**uple **c**entre **t**able

- chop(1)
- chop(4)
- chop(5)
- chop(2, 3)

**b**ehaviour **s**pecification

- reaction(in(chops(C1,C2)), (invocation, operation ), (out(required(C1,C2)))).
- reaction(in(chops(C1,C2)), (completion, operation ), (in(required(C1,C2)))).
- reaction(out(required(C1,C2)), (internal ), (rd(required(C1,C)), in(chop(C1)), in(chop(C)), out(chops(C1,C)))).
- reaction(out(chops(C1,C2)), (completion, operation), (in(chops(C1,C2)), out(chop(C1)), out(chop(C2)))).
- reaction(out(chop(C1)), (internal ), (rd(required(C1,C)), in(chop(C1)), in(chop(C)), out(chops(C1,C)))).
- reaction(out(chop(C2)), (internal ), (rd(required(C2,C)), in(chop(C)), in(chop(C2), out(chops(C,C2))))).

Dining Philosophers: step-by-step inside the TC (4b)

Philo#2 gives chops back

**t**uple **c**entre **t**able

- chop(1)
- chop(2)
- chop(3)
- chop(4)
- chop(5)
- chop(2, 3)

**b**ehaviour **s**pecification

- reaction(in(chops(C1,C2)), (invocation, operation ), (out(required(C1,C2)))).
- reaction(in(chops(C1,C2)), (completion, operation ), (in(required(C1,C2)))).
- reaction(out(required(C1,C2)), (internal ), (rd(required(C1,C)), in(chop(C1)), in(chop(C)), out(chops(C1,C)))).
- reaction(out(chops(C1,C2)), (completion, operation), (in(chops(C1,C2)), out(chop(C1)), out(chop(C2)))).
- reaction(out(chop(C1)), (internal ), (rd(required(C1,C)), in(chop(C1)), in(chop(C)), out(chops(C1,C)))).
- reaction(out(chop(C2)), (internal ), (rd(required(C2,C)), in(chop(C)), in(chop(C2), out(chops(C,C2))))).
Dining Philosophers:
step-by-step inside the TC (4c)

**Philo#2 gives chops back**

reaction(in(chops(C1,C2)), (invocation, operation ), (out(required(C1,C2)))).
reaction(in(chops(C1,C2)), (completion, operation ), (in(required(C1,C2)))).
reaction(out(required(C1,C2)), (internal ), (in(chop(C1)), in(chop(C2)), out(chops(C1,C2)))).
reaction(out(chops(C1,C2)), (completion, operation), (in(chops(C1,C2)), out(chop(C1)), out(chop(C2)))).
reaction(out(chop(C1)), (internal ), (rd(required(C,C2)), in(chop(C1)), in(chop(C2)), out(chops(C,C2)))).
reaction(out(chop(C2)), (internal ), (rd(required(C,C2)), in(chop(C)), in(chop(C2)), out(chops(C,C2)))).

These reactions are both triggered, but this time both fail since there are neither required(_,2) nor required(3,_) tuples in the space – i.e., neither philo#1 nor philo#3 are waiting.

Dining Philosophers:
step-by-step inside the TC (5)

**Philo#2 thinking...**

reaction(in(chops(C1,C2)), (invocation, operation ), (out(required(C1,C2)))).
reaction(in(chops(C1,C2)), (completion, operation ), (in(required(C1,C2)))).
reaction(out(required(C1,C2)), (internal ), (in(chop(C1)), in(chop(C2)), out(chops(C1,C2)))).
reaction(out(chops(C1,C2)), (completion, operation), (in(chops(C1,C2)), out(chop(C1)), out(chop(C2)))).
reaction(out(chop(C1)), (internal ), (rd(required(C,C1)), in(chop(C1)), in(chop(C)), out(chops(C,C1)))).
reaction(out(chop(C2)), (internal ), (rd(required(C,C2)), in(chop(C1)), in(chop(C2)), out(chops(C,C2)))).

Enrico Denti (Università di Bologna) Coordination Models, Languages and Infrastructures Bologna, 2/7/2010
Dining Philosophers: Let’s run (1/2)
using the online console

First, the table tuple centre must be created, and the corresponding specifications loaded.

Then, the set of chop(\text{i}) tuples must be added, by means of a series of \text{out(chop(\text{i}))} operations.
To check that they have been actually added, a series of \text{rdp(chop(\text{i}))} may be performed.

For maximum transparency, it can also be checked that no chop(\text{i, j}) tuples are present.
We are now ready to start playing..!

Dining Philosophers: Let’s run (2/2)
using the online console

Philo#2 wants to start eating: to this end, she asks for her chopstick pair by performing an \text{in(chops(2,3))}
As expected, the reactions generate such a tuple, allowing the \text{in} operation to be successfully served.

To verify that the reaction worked correctly, we can check for the presence of the single chop(2) and chop(3) tuples, none of which is in the space.
This is a stable state, with Philo#2 eating.

When Philo#2 eventually gives her chopstick pair back, she performs an \text{out(chops(2,3))} operation.
However, the associated reaction removes such a tuple immediately, as shown by the failure of \text{rd(chops(2,3))}.
Moreover, the single chop(2) and chop(3) tuples have re-appeared in the space, as shown by the rdp operations.
Remember? The requirements of tuple centres included the support for multiple spaces, also accounting for tuple space distribution over a network.

In addition, the re-interpretation of tuple centres as specialised coordination artifacts highlights some crucial desirable properties:

- **inspectability & controllability**
  - that is, observing & controlling the tuple space structure, state and behaviour
  - at least, for monitoring and debugging purposes
  - ReSpecT tuple centres are inspectable (but not controllable)

- **malleability & forgeability**
  - that is, adapting & changing the tuple space function, state and behaviour
  - nice for incremental development, but also for run-time adaptation & change
  - ReSpecT tuple centres are malleable – just change the behaviour specification

- **linkability & distribution**
  - that is, composing distributed tuple spaces
  - desirable for separation of concerns, encapsulation & scalability
  - partial support in ReSpecT 1.x, full support in ReSpecT 2.x

- **situatedness (in space & time)**
  - that is, be sensitive to the “surrounding” environment – including time
  - implies reacting to other events rather than invocations of coordination primitives
  - supported in ReSpecT 2.x only

By developers, via the ReSpecT Inspector tool:
- inspection: through a coordination primitive
- by agents, via the ReSpecT primitives:
  - for the tuple space: in, out
  - for the specification space: in_s, out_s

By agents, via the ReSpecT primitives:
- malleability & forgeability:
  - that is, adapting & changing the tuple space
  - nice for incremental development, but also
  - ReSpecT tuple centres are malleable

- linkability & distribution:
  - that is, composing distributed tuple spaces
  - desirable for separation of concerns, encapsulation & scalability
  - partial support in ReSpecT 1.x, full support in ReSpecT 2.x

- situatedness (in space & time):
  - that is, be sensitive to the “surrounding” environment – including time
  - implies reacting to other events rather than invocations of coordination primitives
  - supported in ReSpecT 2.x only

- TC are immersed in time
  - (Omicini & al, 2007)

- TC are immersed in environment
  - (Casadei & Omicini, 2009)
Linkability in action: Distributed Dining Philosophers (1)

- The change of perspective
  - philosophers are distributed on the network
- The change in structure & representation
  - each philosopher is assigned a tuple centre which represents his/her seat
    - an individual artifact
  - convention: philo#K \rightarrow seat(K) tuple centre
  - the table tuple centre (the social artifact) with the chop(1) tuples remains untouched along with the "ratio" of the coordination laws
- Agent’s protocol
  - philosophers now act on the table indirectly, via their seat(K) tuple centre (linkability)
  - so, they no longer get/release chops(i,j) like before – they delegate this to their seat
  - rather, each philosopher simply express his intention to eat / think via a wanna_eat / wanna_think tuple, respectively, to be emitted into its seat tuple centre.

Going further: Distributed Dining Philosophers (2)

Previous table coordination laws (concentrated context)

<table>
<thead>
<tr>
<th>Reaction</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>reaction(in(chops(C1,C2)), (invocation, operation), (out(required(C1,C2))))</td>
<td></td>
</tr>
<tr>
<td>reaction(in(chops(C1,C2)), (completion, operation), (in(required(C1,C2))))</td>
<td></td>
</tr>
<tr>
<td>reaction(out(required(C1,C2)), (internal), (in(chop(C1)), (in(chop(C2)), out(chops(C1,C2))))</td>
<td></td>
</tr>
<tr>
<td>reaction(out(chops(C1,C2)), (completion, operation), (in(chops(C1,C2)), out(chop(C1)), out(chop(C2))))</td>
<td></td>
</tr>
<tr>
<td>reaction(out(chop(C1)), (internal), (rd(required(C1,C)), in(chop(C1)), in(chop(C)), out(chops(C1,C))))</td>
<td></td>
</tr>
<tr>
<td>reaction(out(chop(C2)), (internal), (rd(required(C,C2)), in(chop(C)), in(chop(C2)), out(chops(C,C2))))</td>
<td></td>
</tr>
</tbody>
</table>

New table coordination laws (distributed context)

<table>
<thead>
<tr>
<th>Reaction</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>reaction(in(chops(C1,C2)), (invocation, link_in), (out(required(C1,C2))))</td>
<td></td>
</tr>
<tr>
<td>reaction(in(chops(C1,C2)), (completion, link_in), (in(required(C1,C2))))</td>
<td></td>
</tr>
<tr>
<td>reaction(out(required(C1,C2)), (internal), (in(chop(C1)), (in(chop(C2)), out(chops(C1,C2))))</td>
<td></td>
</tr>
<tr>
<td>reaction(out(chops(C1,C2)), (completion, link_in), (in(chops(C1,C2)), out(chop(C1)), out(chop(C2))))</td>
<td></td>
</tr>
<tr>
<td>reaction(out(chop(C1)), (internal), (rd(required(C1,C)), in(chop(C1)), in(chop(C)), out(chops(C1,C))))</td>
<td></td>
</tr>
<tr>
<td>reaction(out(chop(C2)), (internal), (rd(required(C,C2)), in(chop(C)), in(chop(C2)), out(chops(C,C2))))</td>
<td></td>
</tr>
</tbody>
</table>
Going further: Distributed Dining Philosophers (3)

- **Philosopher / Seat interaction**
  - $\text{seat}(i)$ maintains the state of its philosopher—one of thinking, eating, waiting to eat, waiting to think—as suitable tuples
  - when the philosopher is eating, the $\text{chops}(i,j)$ tuple, previously got by the agent directly, is now hosted in the seat $\text{TC}$

- **State transitions only occur when they are safe**
  - from waiting to think to thinking only when chopsticks are safely back on the table
  - from waiting to eat to eating only when chopsticks are actually at the seat $\text{TC}$

```prolog
definitions:
  reaction(out(wanna_eat), (operation, invocation), { current_target(seat([C]), table@node ? in(chops([C],[C2])) ) }).
definitions:
  reaction(out(wanna_eat), (operation, completion), in(wanna_eat)).
definitions:
  reaction(in(chops([C1],[C2])), (link_out, completion), { in(philosopher(waiting_to_eat)), out(chops([C1],[C2])) }).
definitions:
  reaction(out(wanna_think), (operation, invocation), { current_target(seat([C]), in(chops([C1],[C2])), table@node ? out(chops([C1],[C2])) ) }).
definitions:
  reaction(out(wanna_think), (operation, completion), out(wanna_think)).
definitions:
```

Situated ReSpecT

- **Immersion in time**
  - reacting to time events
    - $\text{reaction}(\text{time}(T), [Guard], \text{Body})$
    - a reaction triggered only once, when time reaches $T$—and is discarded after that.
  - observing time properties of events
  - implementing timed coordination policies

- **Immersion in the environment**
  - reacting to environment events
    - ability to capture general environment events,
    - so as to generally mediate agent-environment interaction
  - observing environmental properties
    - ReSpecT 2.x can capture environment events, and express general MAS/environment interactions
    - it can capture, react to, and observe general environment events (including time)
  - affecting environmental properties
    - ReSpecT 2.x can explicitly interact with the environment
Time situatedness in action: Timed Philosophers (1)

- The change of perspective
  - philosophers cannot keep chopsticks forever
  - a timeout is in force: when it expires, chopstick release is forced

- Agent’s protocol
  - no change at all

- The change in coordination laws
  - when the chops(i, j) is given to the agent, its timeout is computed, stored as a has_chops(i, j, from(time)) tuple and used to set up a timed reaction
  - when the timeout expires, the timed reaction:
    - does nothing (i.e. fails) if chopsticks have already been released
    - releases chopsticks forcibly, otherwise

- Let us start from the original (concentrated) coordination laws...
- ...and first just make a "cosmetic" change (propaedeutic to the addition of the timed reaction)

Time situatedness in action: Timed Philosophers (2)

Previous table coordination laws (concentrated context)

<table>
<thead>
<tr>
<th>Reaction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reaction(in(chops(C1,C2)), (invocation, operation), (out(required(C1,C2))))</td>
<td></td>
</tr>
<tr>
<td>reaction(out(required(C1,C2)), (internal), (in(chop(C1)), in(chop(C2)), out(chops(C1,C2))))</td>
<td></td>
</tr>
</tbody>
</table>

Previous table coordination laws (with a cosmetic change)

<table>
<thead>
<tr>
<th>Reaction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reaction(in(chops(C1,C2)), (invocation, operation), (out(required(C1,C2))))</td>
<td></td>
</tr>
<tr>
<td>reaction(out(required(C1,C2)), (internal), (in(chop(C1)), in(chop(C2)), out(chops(C1,C2))))</td>
<td></td>
</tr>
<tr>
<td>reaction(out(chops(C1,C2)), (completion, operation), (in(chop(C1)), out(chop(C1)), out(chop(C2))))</td>
<td></td>
</tr>
<tr>
<td>reaction(out(chop(C1)), (internal), (rd(required(C1,C)), in(chop(C1)), in(chop(C)), out(chops(C1,C))))</td>
<td></td>
</tr>
<tr>
<td>reaction(out(chop(C2)), (internal), (rd(required(C,C2)), in(chop(C)), in(chop(C2)), out(chops(C,C2))))</td>
<td></td>
</tr>
</tbody>
</table>
**Time situatedness in action:**

**Timed Philosophers (3)**

**Previous table coordination laws (with the cosmetic change)**

<table>
<thead>
<tr>
<th>Reaction</th>
<th>Coordination</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>reaction (in(chops(C1,C2)), (invocation, operation), [out(required(C1,C2))]).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reaction (in(chops(C1,C2)), (completion, operation), [in(required(C1,C2))]).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reaction (out(required(C1,C2)), [internal], [in(chop(C1)), in(chop(C2)), out(chops(C1,C2))]).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reaction (out(chops(C1,C2)), (completion, operation), [in(chops(C1,C2))]).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reaction (out(chops(C1,C2)), (completion, operation), [out(chop(C1)), out(chop(C2))]).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reaction (out(chop(C1)), (internal), [rd(required(C1,C2)), in(chop(C1)), in(chop(C2)), out(chops(C1,C2))]).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reaction (out(chop(C2)), (internal), [rd(required(C1,C2)), in(chop(C1)), in(chop(C2)), out(chops(C1,C2))]).</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Normal termination:** clean up the space, removing also the timeout stored

**New table coordination laws (step 1 of 2 – still missing the timed reaction)**

<table>
<thead>
<tr>
<th>Reaction</th>
<th>Coordination</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>reaction (in(chops(C1,C2)), (invocation, operation), [out(required(C1,C2))]).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reaction (in(chops(C1,C2)), (completion, operation), [in(required(C1,C2))]).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reaction (out(required(C1,C2)), [internal], [in(chop(C1)), in(chop(C2)), out(chops(C1,C2))]).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reaction (out(chops(C1,C2)), (completion, operation), [in(chops(C1,C2))]).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reaction (out(chops(C1,C2)), (completion, operation), [out(chop(C1)), out(chop(C2))]).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reaction (out(chop(C1)), (internal), [rd(required(C1,C2)), in(chop(C1)), in(chop(C2)), out(chops(C1,C2))]).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reaction (out(chop(C2)), (internal), [rd(required(C1,C2)), in(chop(C1)), in(chop(C2)), out(chops(C1,C2))]).</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Time situatedness in action:**

**Timed Philosophers (4)**

**New table coordination laws (step 1 of 2 – still missing the timed reaction)**

<table>
<thead>
<tr>
<th>Reaction</th>
<th>Coordination</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>reaction (in(chops(C1,C2)), (invocation, operation), [out(required(C1,C2))]).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reaction (in(chops(C1,C2)), (completion, operation), [in(required(C1,C2))]).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reaction (out(required(C1,C2)), [internal], [in(chop(C1)), in(chop(C2)), out(chops(C1,C2))]).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reaction (out(chops(C1,C2)), (completion, operation), [in(chops(C1,C2))]).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reaction (out(chops(C1,C2)), (completion, operation), [out(chop(C1)), out(chop(C2))]).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reaction (out(chop(C1)), (internal), [rd(required(C1,C2)), in(chop(C1)), in(chop(C2)), out(chops(C1,C2))]).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reaction (out(chop(C2)), (internal), [rd(required(C1,C2)), in(chop(C1)), in(chop(C2)), out(chops(C1,C2))]).</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Time reaction:** when time(T1) is reached, if the agent still has chops, the release is forced.

**New table coordination laws (step 2 of 2 – the new timed reaction)**

<table>
<thead>
<tr>
<th>Reaction</th>
<th>Coordination</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>reaction (in(chops(C1,C2)), (operation, completion), [current_time(T), rd(max_eating_time(TMax))], T1 is T + TMax, out(has_chops(C1,C2,from(T))), out_s(reacton(time(T1), [in(has_chops(C1,C2,from(T))), out(chop(C1)), out(chop(C2))])]).</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTICE:** like all timed reaction, it can be triggered only once – when time reaches T1. So, it is automatically discarded after use.
Some (convincing 😊) examples

From Enabling Communication
to Enabling Interoperability
to Enabling & Performing Coordination

Enabling Communication
(pure generative, Linda-like communication)

Message Passing

![Message Passing Diagram]

<table>
<thead>
<tr>
<th>SENDER:</th>
<th>RECEIVER (entityB):</th>
</tr>
</thead>
<tbody>
<tr>
<td>out(msg(entityB, content('test', 13)))</td>
<td>in(msg(entityB, Info))</td>
</tr>
</tbody>
</table>

Service-Oriented Interaction (RPC style)

<table>
<thead>
<tr>
<th>SERVICE USER:</th>
<th>SERVICE PROVIDER:</th>
</tr>
</thead>
<tbody>
<tr>
<td>out(compute_sum(5, 8, for(henry)))</td>
<td>in(compute_sum(X, Y, for(Who)))</td>
</tr>
<tr>
<td>in(get_sum_result(for(henry), Value))</td>
<td>Sum ➞ X + Y</td>
</tr>
<tr>
<td>out(get_sum_result(for(Who), Sum))</td>
<td></td>
</tr>
</tbody>
</table>
Enabling Communication
(pure generative, Linda-like communication)

Synchronous communication

SYNC MESSAGE SENDER
in(get)
out(msg(...) )
in(got)

SYNC MESSAGE RECEIVER
out(get)
in(Message)
out(got)

• Only when the receiver emits the get tuple, the sender can actually produce its message. Then, it suspends again.
• Only when the receiver emits the got tuple, signaling that it actually got the message, the sender unblocks and proceeds.

Object method invocation

- The client asynchronously emits its "call message"
- The (receiver) object waits for an incoming request and behaves accordingly (possibly changing its state)

OBJECT CLIENT:
out(msg(counterB,inc(13)))

OBJECT (counterB) [count is a private field..]
in(msg(counterB,Method)
if (Method==inc(X)){
count  count + X
}

One step further: Enabling Interoperability

Mediating different ontologies

SERVICE USER:
out(compute_sum(5,8,for(henry)))
in(get_sum_result(for(henry),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))
out(make_sum(term(X,Y)))
in(sum_result(X,Y,Sum))
in(service_requested(sum(X,Y),Who))
out(get_sum_result(Who,Sum))
One step further: Enabling Interoperability via a Tuple Centre

Mediating different ontologies

SERVICE USER:

\[
\begin{align*}
\text{out}(\text{compute}_\text{sum}(5, 8, \text{for(henry)})) \\
\text{in}(\text{get}_\text{result}(\text{for(henry)}, \text{Value}))
\end{align*}
\]

SERVICE PROVIDER

\[
\begin{align*}
\text{in}(\text{make}_\text{sum}(\text{term}(X, Y))) \\
\text{Sum} \leftarrow X + Y \\
\text{out}(\text{sum}_\text{result}(X, Y, \text{Sum}))
\end{align*}
\]

The ontology mediation, needed for interoperability, is now charged onto the coordination medium.

TUPLE CENTRE as a MEDIATOR

\[
\text{reaction(out(compute}_\text{sum}(X,Y, \text{Who})), \text{completion}, \{
\text{in(compute}_\text{sum}(X,Y, \text{Who}))},
\text{out(service}_\text{requested(sum}(X,Y), \text{Who})),
\text{out(make}_\text{sum}(\text{term}(X,Y)) \})
\]

\[
\text{reaction(out(sum}_\text{result}(X,Y, \text{Sum})), \text{completion}, \{
\text{in(sum}_\text{result}(X,Y, \text{Sum})),
\text{in(service}_\text{requested(sum}(X,Y), \text{Who})),
\text{out(get}_\text{result}(\text{Who}, \text{Sum})) \})
\]

Coordination Patterns in Linda & ReSpecT

Basic Synchronisation (on a clock)

A global clock

CLOCK component:

\[
\begin{align*}
\text{Loop forever} \\
\text{in(clock}(C)) \\
\text{Cl} \leftarrow C + 1 \\
\text{out(clock}(\text{Cl})) \\
\end{align*}
\]

USER:

\[
\text{rd(clock}(X))
\]

The CLOCK component is charged of handling the actual state (other than counting time...)

\(\Rightarrow\) undesired coupling

TUPLE CENTRE charged of the coordination policy

\[
\text{reaction(out(tick)), completion, \{
\text{in(tick)},
\text{in(clock}(N)),
N1 \text{ is } N+1,
\text{out(clock}(N1)) \})
\]

The new perspective: CLOCK just counts time
Handling the rest is NOT its task \(\Rightarrow\) full uncoupling

\(\text{(The user is unchanged)}\)
Coordination Patterns in Linda & ReSpecT
Barrier Synchronisation

Barrier Synchronisation

COORDINABLE A:
<before barrier>
  out(ready(entityA))
  rd(ready(entityB))
  <A and B are synchronised>
...

COORDINABLE B:
<before barrier>
  out(ready(entityB))
  rd(ready(entityA))
  <A and B are synchronised>
...

Nice, but...
- Adding an entity implies changing all the others
- Every entity must be aware of all the others
- Definitely, not scalable!

COORDINABLE A:
<before barrier>
  out(ready(entityB))
  rd(ready(entityA))
  rd(ready(entityC))
  <A,B,C are synchronised>
...

COORDINABLE B:
Nice, but...
- Adding an entity implies changing all the others
- Every entity must be aware of all the others
- Definitely, not scalable!

Barrier Synchronisation via a Tuple Centre

Barrier Synchronisation

HYPOTHESIS:
Initially, the tuple space contains
  • barrier_size(N) [number of entities]
  • ready_entities(0) [initial barrier state]

TUPLE CENTRE COORDINATOR
reaction(out(ready), (operation, completion),
in(ready),
in(ready_entities(N)))), M1 is N+1,
out(ready_entities(N1)))
)
reaction(out(ready_entities(N)), internal, {rd(barrier_size(N)), out(ready_all))})

Scalable approach:
- all coordinables are identical...
- and even much simpler!

EACH COORDINABLE:
<before barrier>
  out(ready)
  rd(ready_all)
  <ALL are synchronised>
...

No more entity name
Increments the number of ready entities...
...until all are ready
Coordination Patterns in Linda & ReSpecT
Resource Sharing/Allocation

A dynamic/open set of users accessing a resource according to a given coordination policy (e.g.: FIFO)

Each coordinable must handle tickets and turns to manage synchron. & coord. policy
- undesired coupling
- critical section (non atomic)
- FIFO policy embedded in the coordinable

Initial HYPOTHESIS:
- next_ticket(0)
- current_turn(0)

What if...
- we need to change the coordination policy?
- a user fails or is malicious?
  - not flexible & dangerous.

EACH COORDINABLE:

in(next_ticket(T))
T1 ← T + 1
out(next_ticket(T1))
in(current_turn(T))
<use the resource>
out(current_turn(T1))

1. Encapsulating the (FIFO) sharing policy

reaction(in(resource_token(Who)), {operation, invocation}, {in(next_ticket(N)), N1 is N + 1, out(next_ticket(N1)), out(has_reservation(Who,N))}).
reaction(out(has_reservation(Who,N)), internal, {rd(current_turn(N)), out(resource_token(Who))}).
reaction(out(resource_token(Who)), {operation, completion}, {in(has_reservation(Who,N)), in(current_turn(N)), N1 is N+1, out(current_turn(N1))}).
reaction(out(current_turn(N)), internal, {rd(has_reservation(Who,N)), ? ? out(resource_token(Who))}).

- when an entity tries to get access to the resource, its request reservation is registered with its turn
- if it’s its turn (i.e. its ticket number = the current turn), resource access is granted
- otherwise, the reaction just fails, yielding no effect

- when a new turn comes, a check is made to see if there are any pending reservations for it
- if so, resource access is granted to the waiting entity.

FIFO policy

resource access is granted to the waiting entity.

- when the entity releases the resource, its request reservation is cancelled, and the next turn is started.
Coordination Patterns in Linda & ReSpecT
Resource Sharing/Allocation via a Tuple Centre

2. ..and easily changing from FIFO to LIFO!

reaction(in(resource_token(Who)), (operation, invocation), (in(next_ticket(N)), N1 is N + 1, out(has_reservation(Who, N)))).
reaction(out(has_reservation(Who, N)), internal, {rd(current_turn(N)), out(resource_token(Who))}).
reaction(out(resource_token(Who)), (operation, completion), (in(resource_token(Who)), in(has_reservation(Who, N)), in(current_turn(N)), rd(next_ticket(LastN)), N1 is LastN-1, out(current_turn(N1)))).
reaction(out(current_turn(N)), internal, {rd(has_reservation(Who, N)), no(has_reservation(Who, N)), N1 is N-1, N1>=0, in(current_turn(N)), out(current_turn(N1))}).
reaction(out(current_turn(0)), internal, {rd(has_reservation(Who, 0)), no(has_reservation(Who, 0)), in(next_ticket(_)), out(next_ticket(0))}).

Initial HYPOTHESIS:
• next_ticket(0)
• current_turn(0)
• when the resource is released, the next turn is NOT the next, but the LAST arrived
• however, the last might have already been served
• but other, previous requests might still be waiting...
• so, previous turns are checked
• the highest with a reservation is served

2. ..and easily changing from FIFO to LIFO!

ReSpecT summing up: main benefits...

- Impact on the observation & construction of the interaction space
  - Direct support for static / dynamic communication & coordination state

- Impact on design
  - Encapsulation of coordination policies (objective coordination)
    - full separation between computation and coordination
    - reuse of coordination patterns embodied in coordination media
  - Chance to balance the burden of coordination between the coordinables and the coordination media, on a by-need base

- Impact on coordination scalability
  - Reduce the number of (unnecessary) interactions
  - potential positive impact on performance

- Impact on openness
  - Support for heterogeneous coordinables
  - Support for openness – coordinables can be added/removes from system
  - Dynamic change/adaptation of coordination policies
.. and some reflections

- ReSpecT is powerful, yet still somehow low level
  - a sort of "assembly language" of interaction/coordination
    - PROs:
      - general-purpose enough to support potentially any coordination policy
      - formal semantics
    - CONs:
      - it may be hard to specify complex coordination patterns
      - higher-level languages, frameworks possibly required...

- Ongoing research directions
  - to map higher level specification languages on top of ReSpecT
    - BPEL (for workflow management)
    - Petri Nets (to take advantage of tools & properties)
  - to further explore the situatedness dimension
    - relation with the environment

In conclusion..

- Computation is only a part of the story
  - Interaction & Coordination as first-class conceptual and engineering dimension requiring models, methodologies, languages, tools

- Coordination Models and Languages address interaction & coordination as first-class issues
  - proper abstraction level for interaction/coordination complexity

- Generative communication
  - makes coordination easy & uniform …
  - …but still puts lots of coordination burden on the coordinables

- Encapsulating coordination
  - opens the way to the engineering of coordination
    - reusable coordination artifacts, embedding general-purpose patterns
    - decide the balancing of the coordination burden between coordinables and media as a design choice, rather than a constraint
    - the base for building coordination infrastructures
Towards Coordination Infrastructures

The LuCe & TuCSoN Coordination Infrastructures

**LuCE** *(Logic tuple CENTres)*

- **The model** [Denti & Omicini, 1999, 2001]
  - based on tuple centres
  - tuple centres spread over *cells*
  - intentionally no notion of topology
  - *coordination-transparent*
    - tuple centres referenced by name wherever they are (*“in the cloud”*)

- **The infrastructure**
  - Client / server architecture
    - Java-based
    - simple *GUI shell* for interaction
    - Prolog & Java agents only
  - Support tools
    - a *simple Inspector* to help inspect/monitor/debug tuple centres dynamics

**TuCSoN** *(Tuple CENTres Spread over the Network)*

- **The model** [Omicini & Zambonelli, 1999]
  - based on tuple centres
  - tuple centres spread over *nodes*
  - each node defines a *coordination context*
  - topology and *coordination awareness*
  - organisation & security issues
    - Agent Coordination Contexts (ACC)

- **The infrastructure**
  - Client / server architecture
    - a *service daemon* for the TuCSoN node
    - a *TuCSoN shell* for interaction
    - *TuCSoN API* for agent support
  - Support tools
    - a full-fledged *Inspector tool* to inspect / monitor/debug tuple centres dynamics
    - experimental: the *Web-based TuCSoN*
The TuCSoN model

The TuCSoN model

- **Distributed** coordination media: tuple centres are spread on network **nodes**
  - TuCSoN nodes are Internet nodes, identified by their IP (or logic) addresses
- Each node defines a **coordination context**, which can host **multiple tuple centres**
  - The set of tuple centres hosted in a node is NOT fixed or pre-defined
  - nor is it limited (unless by physical resources) → open & dynamic set of tuple centres
- Tuple centres naming conventions:
  - each **tuple centre name** is a logic ground term
  - room('2.3'), seat(3,4), mail(aricci), ...
  - the **full tuple centre identifier** includes the node with the classical @node notation
  - room('2.3')@unibo.it, seat(3,4)@ed.org, ...
- **Agent's access** to a coordination context
  - to access a TuCSoN coordination context, agents must negotiate their entrance (rights, etc)
  - concept of **Agent Coordination Context (ACC)** [Omicini, 2001]

Agent Coordination Contexts (ACC) (1)

**Motivations in short**
- In an open world, components need some form of **context awareness** to interact with both other agents and the environment
- An infrastructure for open worlds should then support not only inter-agent interactions, but also agent-environment interactions

**ACC basics**
- The basic idea: a sort of **control room** providing agents with context awareness
  - The (only) way in which an agent can perceive the environment and interact
  - but also a model for the agent environment, describing the environment where agents live
- Two stages: **negotiation** and **dynamics**
  - first, agents negotiate with the infrastructure the desired properties (role) for their ACC
  - then, agents use it to interact with the organisation & environment, performing the actions and getting the perceptions enabled&granted by the ACC
- Not only an abstraction for human designers, but a run-time service available to agents

**ACC framework as an orthogonal dimension**
- to the computational model(s) used for defining agent behaviour
- to the interaction model(s) adopted to specify how agents communicate and interact within the environment
The TuCSoN model
Agent Coordination Contexts (ACC) (2)

**The Subjective viewpoint: ACC**
- ACC provide agents with a suitable representation of the environment where they live, interact and communicate.
- ACC enables agents to perceive the space where they act and interact, reason on the effect of their actions and communications, and possibly affect the environment to accomplish their own goals.

**The Objective viewpoint: TC**
- TCs provide a framework to express the interaction within a MAS as a whole, i.e. the space of MAS interaction.
- TC encapsulate the rules to govern agent systems, mediate the interactions among agents and environment, and possibly affect them so as to change the global application behaviour incrementally and dynamically.

**ACC general model**
- **ACC Interface**: defines the set of admissible operations provided by the infrastructure for interacting with the (social and resource) environment
- **ACC Contract**: describes the agent-environment relationships
  - in particular, the policy enacted by the ACC to rule agent actions and interaction protocols
- **ACC State**: describes the run-time evolution of the ongoing interaction protocols

**ACC as a unifying modelling abstraction** [Ricci et al, 2006]
- a unifying abstraction for modelling heterogeneous issues in MAS engineering
  - namely, organisation, access control, quality of interaction, relation between agents & institutions

---

**TuCSoN technology: from the origins to... the future**


- **TuCSoN 1st version (1.3.0) [2002]**
  - architecture (node, agents) & tools (inspector)
  - ReSpecT engine 1.4
  - no support for persistency, nor for ACCs
- **TuCSoN 2nd version (1.4.0) [2004]**
  - same architecture, improved tools
  - ReSpecT engine 1.5
  - persistency supported, but still no support for ACCs
- **TuCSoN 3rd version (1.4.5) [2006]**
  - basic support for ACCs (agent skeletons & node stubs)
  - but still no support for policies inside ACCs
    - (only two roles – inspector/agent – distinguished)
- **TuCSoN 4th version (1.9.1) [2009]**
  - refactored version, separated ReSpecT engine
  - ReSpecT engine 2.2
- **TuCSoN 5th version (1.9.2) [3rd quarter 2010]**
  - embedding new tuProlog engine
  - semantic TCs: associated ontology + reasoner + semantic match

---

**TuCSoN node as WinXP service (.NET)**
- currently under update...

**TuCSoN as a Web Service**
- future redesign...
The TuCSoN technology:
Node and Shell

The TuCSoN node (version 1.4.5)
java -cp lib/tucson.
       -jar alice.tucson.service.Node
Things to note:
- organisation tuple centre ("$ORG")
- persistency
- "default allow" policy for all roles

The TuCSoN shell (version 1.4.5)
java -cp lib/tucson.
       -jar alice.tucson.tools.Shell
Things to note:
- organisation tuple centre ("$ORG")
- persistency
- "default allow" policy for all roles

The TuCSoN technology—Mainstream
The Inspector tool (1)

The TuCSoN Inspector (version 1.4.5)
java -cp lib/tucson.
       -jar alice.tucson.tools.Inspector
Clicking on INSPECT, an Inspector console opens on
the specific tuple centre name @ host.

For instance, this is the
Inspector’s console on tuple
centre default @ localhost

The Pending Queries view

The Tuple Space view
The TuCSoN technology – Mainstream

The Inspector tool (2)

The TuCSoN Inspector (version 1.4.5)

del.icio.us/tucson

Clicking on INSPECT, an Inspector console opens on
the specific tuple centre name @ host

The Pending Reaction view

The Specification Space view (editing window)

The TuCSoN technology – Mainstream
Writing (basic) Java agents

A simple Java agent
- connecting to a TuCSoN node
- entering the default context
- using tuple centre test_tc
- performing out(p('hello world'))
- and then an in(p(X)) operation
- whose result is then printed on the
Java console

import alice.tucson.spi.*;
import alice.logictuple.*;

public class Test {
    public static void main(String[] args) throws Exception {
        TuCSoNContext cc = TuCSoN.contextDefaultValueDescription();
        TupleCentreId tc = new TupleCentreId("test_tc");
        cc.out(new Tuple(new LogicTuple("p","hello world")));
        System.out.println(cc.in().toString());
    }
}

Another simple Java agent
- putting in tuple centre temperature
  the tuple value(1,38.5)
- built from a Java string via the
LogicTuple.parse method
- the tuple value is then retrieved and
  printed on the Java console

import logictuple.*;
import tucson.spi.*;

public class Test {
    public static void main(String[] args) throws Exception {
        AgentId agentId="agent-7";
        TuCSoNContext cc = TuCSoN.contextDefaultValueDescription(agentId);
        cc.in().set(new Tuple(new LogicTuple("p",new TupleValue("temperature",1,38.5))));
        System.out.println(cc.in().toString());
    }
}
The TuCSoN technology– Mainstream
Writing Java & tuProlog agents

The TuCSoN API offers the Agent class
- simple support to write agents, overriding the body method (in an Agent subclass)
- then, just create an instance and run the new agent via the spawn method
- a suitable AgentID (a string representing a valid logic term) is needed for that purpose

```java
public class Test {
    public static void main(String[] args) throws Exception {
        AgentID id = new AgentID("alice");
        MyAgent agent = new MyAgent(id);
        System.out.println(agent.getAgentID());
    }
}
```

A simple tuProlog agent
- load the TucsonLibrary (provides the TuCSoN API in a "prolog flavour")
- then, launch the agent's top goal

```prolog
:- load.library('alice.tuprolog.lib.TucsonLibrary').
:- solve(go).

go:-
    test.tc ? out('hello world'),
    test.tc ? in(p(X)),
    write(X), nl.
```

The complete TuCSoN API is described in the Javadoc reference (next slide)
The full TuCSoN manual is also available on the TuCSoN sites.
The TuCSoN technology: side streams

The "Windows XP service Node"

The TuCSoN applet in the Control Panel

Requirement: fully interoperable with TuCSoN Java agents

The "Windows XP service Node"

"Web-based TuCSoN", "TuCSoN as a WS"

TuCSoN as a Web Service
a single Web Service with as many methods as the primitives (out, in, nd, inp, rdp, ..)
technology: JAX-RPC
1. Encapsulating the (FIFO) sharing policy

reaction(in(resource_token(Who)), (operation, invocation), (in(next_ticket(N)), N1 is N + 1, out(has_reservation(Who,N)))).
reaction(out(has_reservation(Who,N)), internal, {rd(current_turn(N)), out(resource_token(Who))}).
reaction(out(resource_token(Who)), (operation, completion), (in(resource_token(Who)), in(has_reservation(Who,N)), in(current_turn(N)), N1 is N+1, out(current_turn(N1)))).
reaction(out(current_turn(N)), internal, {rd(has_reservation(Who,N)), out(resource_token(Who))}).

1. Encapsulating the (FIFO) sharing policy

Initial HYPOTHESIS:
• next_ticket(0)
• current_turn(0)

• when an entity tries to get access to the resource, its request reservation is registered with its turn
• if it’s its turn (i.e. its ticket number is the current turn), resource access is granted
• otherwise, the reaction just fails, yielding no effect

• when the entity releases the resource, its request reservation is cancelled, and the next turn is started
• when a new turn comes, a check is made to see if there are any pending reservations for it
• if so, resource access is granted to the waiting entity.

Back to the Resource Sharing/Allocation example

The FIFO policy case

Resource Sharing/Allocation in ReSpecT & TuCSoN

NOW LIVE ON STAGE!
When John releases the resource, Jenny gets it (FIFO)

Troy remains waiting

John disappeared, two reservations are left
Jenny (#10) is being served
Troy is still waiting (pending request)

Then Jenny releases the resource, and finally Troy too
TC final state below

2. ...and easily changing from FIFO to LIFO!

reaction(in(resource_token(Who)), (operation, invocation), (in(next_ticket(N)), N1 is N + 1, out(next_ticket(N1)), out(has_reservation(Who,N)))).

reaction(out(has_reservation(Who,N)), internal, (rd(current_turn(N)), out(resource_token(Who))).

reaction(out(resource_token(Who)), (operation, completion), (in(resource_token(Who)), in(has_reservation(Who,N)), in(current_turn(N)), rd(next_ticket(LastN)), N1 is LastN-1, out(current_turn(N1))).

reaction(out(current_turn(0)), internal, (no(has_reservation(Who,N)), in(next_ticket(_)), out(next_ticket(0))).

The LIFO policy case
John, Jenny and Troy asked for the resource, in this order.
John got access, the others are waiting.

In fact, the TC state holds 3 reservations, but only John (#0) holds the turn.
Jenny and Troy are waiting (like before).

However, when John releases the resource, Troy (the last!) wins.
Only when Troy releases the resource, Jenny is served.
Finally, the TC state is reset to the initial state.

Case study:
A BPEL Orchestration Engine on the top of TuCSoN.
A BPEL Orchestration Engine on top of TuCSoN (1)

Outside & inside view of a composed service in BPEL4WS (1.1)

BPEL syntax (examples)

Example of Simple Activity

    <invoke partnerLink="ncname" portType="qname" operation="ncname"
      inputVariable="ncname"?
      outputVariable="ncname"?
      standard-attributes
      standard-elements
    </invoke>

Example of Composed Activity

    <switch standard-attributes>
      standard-elements
        <case condition="bool-expr">+
          activity
        </case>
        <otherwise>?
          activity
        </otherwise>
    </switch>

A BPEL Orchestration Engine on top of TuCSoN (2)

Nested Control Flows (nested scopes)

Approach: one scope per tuple centre

The resulting architecture
A BPEL Orchestration Engine on top of TuCSoN (3)

System Architecture

REQUESTS

Ports used by the service

Tuple Centre containing all the information needed to create a process instance

Agent handling one instance of process

Agent programming the TC tuple centre based on the BPEL and .wsdl files

Run-time dynamics

Tuple Centre containing the workflow engine

TC handling instance variables

A BPEL Orchestration Engine on top of TuCSoN (4)

Tuple Centres: workflow engine

The higher-level specification language

Task

Task

link

flow

variable

structure

BPEL legend

<sequence> <switch> <pick> <flow> <while>

Enrico Denti (Università di Bologna) Coordination Models, Languages and Infrastructures Bologna, 2/7/2010
A BPEL Orchestration Engine on top of TuCSoN (5)

Workflow engine implementation (ReSpect 1.x) - page 1 of 2

% Start reaction
% ReSpect 1.x)
\begin{verbatim}
 reaction(out_r(start_scope()),
in_r(start_scope()),
in_r(flow(Name,normal,[StartTask|List],ready),
out_r(flow(Name,normal,[StartTask|List],executing),
in_r(task(TaskName,tag,ready,StartTask,...)
out_r(task(TaskName,tag,executing,StartTask,...)
)
% Completing a task of type "tag"
\begin{verbatim}
 reaction(out_r(task(TaskName,tag,started,StartTask,...)
\end{verbatim}
% Scheduling a task of type "activity"
\begin{verbatim}
 reaction(out_r(task(TaskName,activity,started,StartTask,...)
\end{verbatim}
% Acknowledging the success of a task
\begin{verbatim}
 reaction(out_r(task_success(TaskName,SuccessInfo),
in_r(task(TaskName,activity,started,StartTask,...)
out_r(task(TaskName,activity,executing,StartTask,...)
out_r(update_outgoing_links([Link|List]))
\end{verbatim}
% Updating outgoing links
\begin{verbatim}
 reaction(out_r(update_outgoing_links([Link|List])),
in_r(update_outgoing_links([Link|List])),
rd_r(link(Link,LinkType,Source,Target,_,not_evaluated)),
out_r(link(Link,LinkType,Source,Target,TransitionCondition,evaluated)),
rd_r(task(Target,_,ready,_,_,_,_,_,_,_)),
out_r(verify_incoming_links(Target,[Link|List]))
\end{verbatim}
% Recursive verification of evaluation of incoming links
\begin{verbatim}
 reaction(verify_incoming_links(Target,[Link|List]),
in_r(verify_incoming_links(Target,[Link|List])),
rd_r(link(Link,_,_,_,true,evaluated))
\end{verbatim}
% End of verification - scheduling of task order_evaluation
\begin{verbatim}
 reaction(verify_incoming_links(Target,[Link|List]),
in_r(verify_incoming_links(Target,[Link|List])),
rd_r(task(Target,_,ready,OrderCondition,JoinCondition,_,_,_,_,_)),
out_r(task_to_do(order_evaluation,info(Target,OrderCondition,JoinCondition)))
\end{verbatim}
% Launching a new task
\begin{verbatim}
 reaction(out_r(task_success(order_evaluation,info(OrderRep,JoinRep,Target))),
in_r(task(Target,_,ready,OrderRep,JoinRep,Target),
out_r(task(Target,_,started,OrderRep,JoinRep,Target))
\end{verbatim}
\end{verbatim}

A BPEL Orchestration Engine on top of TuCSoN (5)

Workflow engine implementation (ReSpect 1.x) - page 2 of 2

% Successful evaluation of links triggering verify on targets
\begin{verbatim}
 reaction(out_r(task_success(order_evaluation,info(TaskName,LinkName))),
in_r(task(TaskName,LinkType,Source,Target,TransitionCondition,not_evaluated)),
out_r(link(LinkName,LinkType,Source,Target,TransitionCondition,evaluated)),
rd_r(task(Target,_,ready,_,_,_,_,_,_,_)),
out_r(task_to_do(order_evaluation,info(TaskName,LinkName)))
\end{verbatim}
% Recursive verification of evaluation of incoming links
\begin{verbatim}
 reaction(verify_incoming_links(TaskName,LinkName),
in_r(verify_incoming_links(TaskName,LinkName)),
rd_r(link(Link,_,_,_,true,evaluated))
\end{verbatim}
% End of verification - scheduling of task order_evaluation
\begin{verbatim}
 reaction(verify_incoming_links(TaskName,LinkName),
in_r(verify_incoming_links(TaskName,LinkName)),
rd_r(task(TaskName,_,ready,OrderCondition,JoinCondition,_,_,_,_,_)),
out_r(task_to_do(order_evaluation,info(TaskName,OrderCondition,JoinCondition)))
\end{verbatim}
% Launching a new task
\begin{verbatim}
 reaction(out_r(task_success(order_evaluation,info(TaskName,LinkName))),
in_r(task(TaskName,_,ready,_,_,_,_,_,_,_)),
out_r(task(TaskName,_,started,OrderRep,JoinRep,Target))
\end{verbatim}
TuCSOn Home

TuCSOn (TuCSoN centre over the Network) is a module and related infrastructure and technology for the coordination of Internet agents.

TuCSOn exploits a version of the agent-based infrastructure space, called TuCyber, which is a single open source with the features of the Internet space. By operating on the TuCyber module, it is possible to develop and run computational environments with agents without the need for a specific agent-oriented runtime environment.

TuCyber provides an agent-oriented runtime environment supporting a high-level programming language, an advanced interface for the interaction with other modules, and a set of services for the management of the agents and the environment.

The TuCyber infrastructure, language, and tools provide an open-source environment for the development and deployment of agent-oriented applications on the Internet.

Currently, TuCyber supports the development of agent-based applications in several environments, including the Internet, enterprise networks, and mobile devices.

TuCyber is developed as a modular and extensible system, allowing for the integration of new components and functionalities.

http://tucson.sourceforge.net

TuCyber is a modular and extensible infrastructure enabling the development of agent-oriented applications and services on the Internet.

> more info
> download

http://cartago.apice.unibo.it

CARTAGO (Cartago Architecture for Agent-oriented Programming) is a platform for the development of agent-based systems. CARTAGO provides a modular and extensible architecture for the development of agent-based systems on the Internet.

CARTAGO supports the development of distributed and scalable agent-based systems, allowing for the integration of different components and functionalities.

CARTAGO is designed to be easy to use and to support the development of complex agent-based systems. It provides a set of tools and services for the development and deployment of agent-based applications.

CARTAGO is a modular and extensible platform, allowing for the integration of new components and functionalities.

CARTAGO is a modular and extensible platform, allowing for the integration of new components and functionalities.
Selected Bibliography (1)

Selected Bibliography (2)

  *Coordination models and languages.*

  *On the Expressive Power of a Language for Programming Coordination Media*
  ACM Symposium on Applied Computing (SAC), 1998, ACM

  *Tuple-based Coordination for Multi-Agent Systems*
  Software – Practice & Experience, 29(12), 1999, 1102:1121, John Wiley & Sons

- E. Denti, A. Omicini (2001)
  *LuCe – A Tuple-based Coordination Infrastructure for Prolog and Java*

  *From Tuple Spaces to Tuple Centres*

Selected Bibliography (3)

  *Time-aware coordination in ReSpecT*
  in "Coordination Models and Languages", LNCS 3454, 268-282, Springer, 2005

  *Formal ReSpecT in the A&A Perspective*
  Electronic Notes in Theoretical Computer Science, No. 175, 97-117, 2007

- M. Casadei, A. Omicini (2009)
  *Situated tuple centres in ReSpecT*
  ACM Symposium on Applied Computing (SAC), 2009, ACM