Programming Interaction with ReSpecT

Distributed Systems / Technologies

Sistemi Distribuiti / Tecnologie

Stefano Mariani Andrea Omicini

{s.mariani, andrea.omicini}@unibo.it

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

Alma Mater Studiorum – Università di Bologna a Cesena

Academic Year 2017/2018

Disclaimer

- a good deal of the following slides are adapted from the official TuCSoN guide
- the TuCSoN guide is available at

  `http://www.slideshare.net/andreaomicini/`

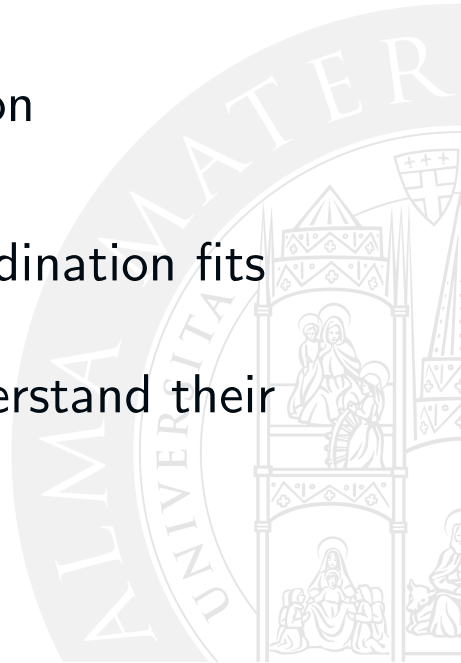
  `the-tucson-coordination-model-technology-a-guide`

Next in Line...

Focus on...

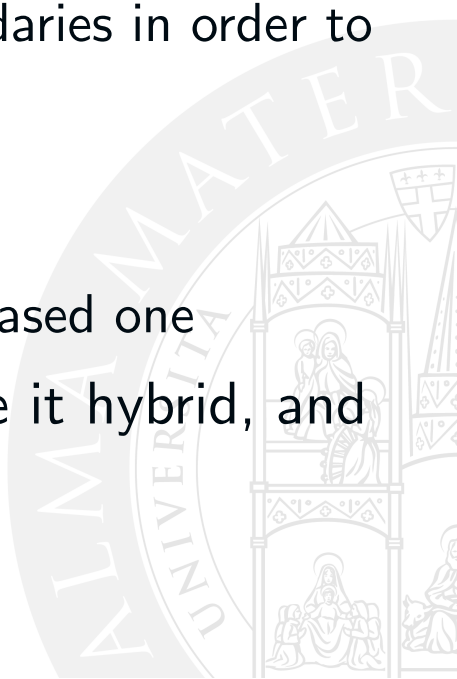
Data- vs. Control-driven Coordination

- what if we need to start an activity after, say, at least N processes have asked for a resource?
  - more generally, what if we need, in general, to coordinate based on the coordinable actions, rather than on the information available / exchanged?
- classical distinction in the coordination community
  - data-driven coordination vs. control-driven coordination
- in more advanced scenario, these names do not fit
  - *information-driven* coordination vs. *action-driven* coordination fits better
  - but we might as well use the old terms, while we understand their limitations



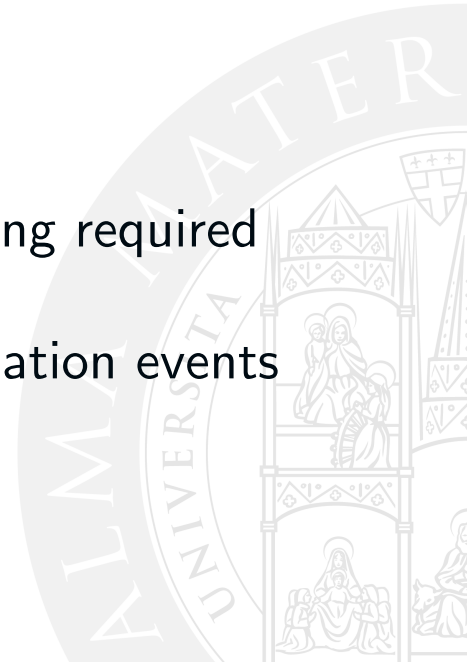
Hybrid Coordination Models

- generally speaking, control-driven coordination does not fit so well information-driven contexts, like Web-based ones, for instance
  - control-driven models like Reo [Arbab, 2004] need to be adapted to contexts like agent-based ones, mainly to deal with the issue of autonomy in distributed systems [Dastani et al., 2005]
  - control should not pass through the component boundaries in order to avoid coupling in distributed systems
- we need features of both approaches to coordination
  - *hybrid* coordination models
  - adding for instance a control-driven layer to a Linda-based one
- what should be added to a tuple-based model to make it hybrid, and how?



Towards Tuple Centres

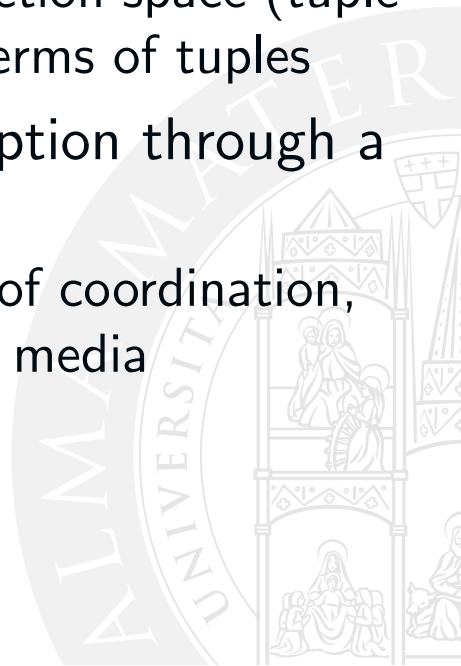
- what should be left unchanged?
  - no new primitives
  - basic LINDA primitives are preserved, both syntax and semantics
  - matching mechanism preserved, still depending on the communication language of choice
  - multiple tuple spaces, flat name space
- new features?
  - ability to define new coordinative behaviours embodying required coordination policies
  - ability to associate coordinative behaviours to coordination events



Focus on...

Ideas from the Dining Philosophers I

1. keeping information representation and perception separated
   - in the tuple space
   - this would enable process interaction protocols to be organised around the desired / required process perception of the interaction space (tuple space), independently of its *actual* representation in terms of tuples
2. properly relating information representation and perception through a suitably defined tuple-space behaviour
   - so, processes could get rid of the unnecessary burden of coordination, by embedding coordination laws into the coordination media



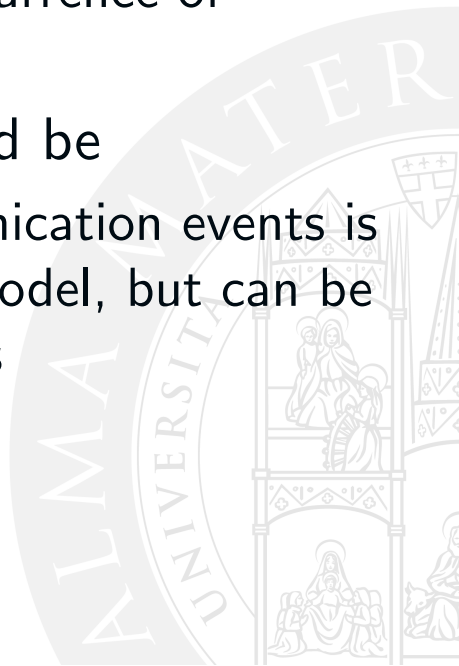
Ideas from the Dining Philosophers II

In the Dining Philosophers example...

- ... this would amount to representing each chopstick as a single `chop(`i`)` tuple in the tuple space, while enabling philosophers to perceive chopsticks as pairs (tuples `chops(`i`,`j`)`), so that philosophers could acquire / release two chopsticks by means of a single tuple space operation `in(chops(`i`,`j`))` / `out(chops(`i`,`j`))`.
- how could we do that, in the example, and in general?

A Possible Solution I

- a twofold solution
  1. maintaining the standard tuple space interface
  2. making it possible to enrich the behaviour of a tuple space in terms of the state transitions performed in response to the occurrence of standard communication events
- so, in principle, the new tuple-based abstraction should be
  - a tuple space whose behaviour in response to communication events is no longer fixed once and for all by the coordination model, but can be defined according to the required coordination policies



A Possible Solution II

Consequences

- since it has exactly the same interface, a tuple centre is perceived by processes as a standard tuple space
- however, since its behaviour can be specified so as to encapsulate the coordination rules governing process interaction, a tuple centre may behave in a completely different way with respect to a tuple space
- we call it a tuple centre

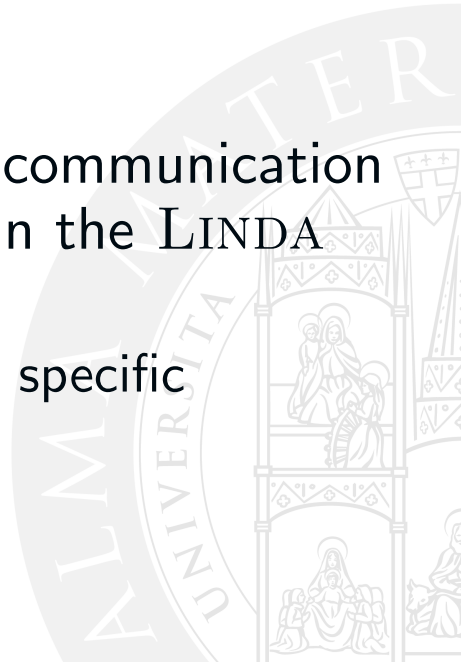
Tuple Centres

Definition

- a tuple centre is a tuple space enhanced with a behaviour specification, defining the behaviour of a tuple centre in response to interaction events [Omicini and Denti, 2001]
- the *behaviour specification* of a tuple centre
  - is expressed in terms of a reaction specification language, and
  - associates any tuple-centre event to a (possibly empty) set of computational activities, which are called reactions
- more precisely, a reaction specification language
  - enables the definitions of computational activities within a tuple centre, called reactions, and
  - makes it possible to associate reactions to the events that occur in a tuple centre

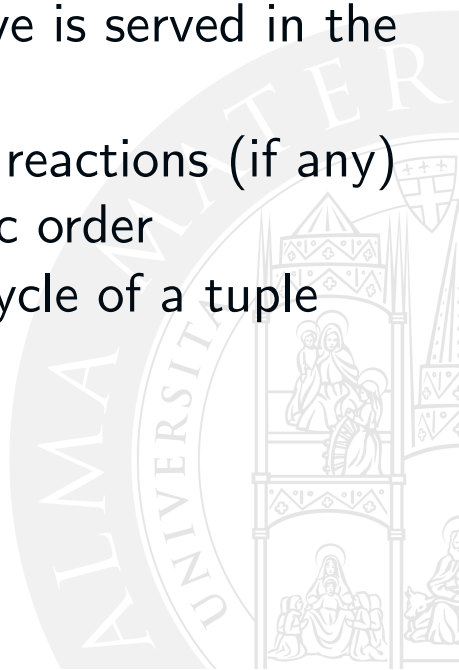
Reactions

- each reaction can in principle
  - access and modify the current tuple centre state—like adding or removing tuples)
  - access the information related to the triggering event—such as the performing process, the primitive invoked, the tuple involved, etc.)—which is made completely observable
  - invoke link primitives upon other tuple centres
- as a result, the semantics of the standard tuple space communication primitives is no longer constrained to be as simple as in the LINDA model—i.e., adding, reading, and removing tuples
  - instead, it can be made as complex as required by the specific application needs



Reaction Execution I

- the main cycle of a tuple centre works as follows
  - when a primitive invocation reaches a tuple centre, all the corresponding reactions (if any) are triggered, and then executed in a non-deterministic order
  - once all the reactions have been executed, the primitive is served in the same way as in standard LINDA
  - upon completion of the invocation, the corresponding reactions (if any) are triggered, and then executed in a non-deterministic order
  - once all the reactions have been executed, the main cycle of a tuple centre may go on possibly serving another invocation



Reaction Execution II

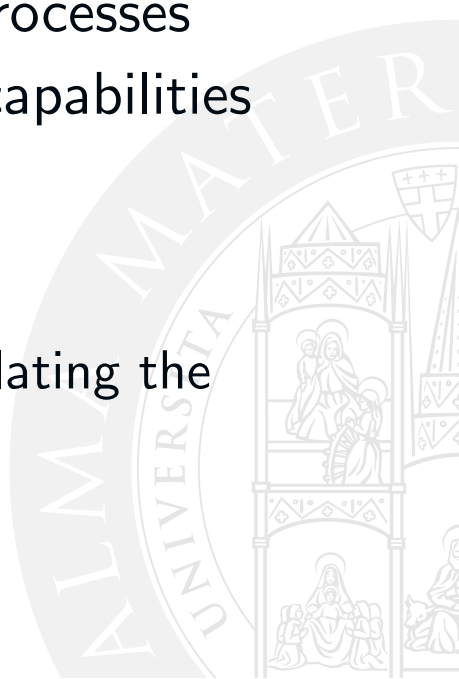
- as a result, tuple centres exhibit a couple of fundamental features
  - since an empty behaviour specification brings no triggered reactions independently of the invocation, the behaviour of a tuple centre defaults to a tuple space when no behaviour specification is given
  - from the process's viewpoint, the result of the invocation of a tuple centre primitive is the sum of the effects of the primitive itself and of all the reactions it triggers, perceived altogether as a single-step transition of the tuple centre state

Tuple Centre State vs. Process Perception

- reactions are executed in such a way that the observable behaviour of a tuple centre in response to a communication event is still perceived by processes as a single-step transition of the tuple-centre state
  - as in the case of tuple spaces
  - so tuple centres are perceived as tuple spaces by processes
- unlike a standard tuple space, whose state transitions are constrained to adding, reading or deleting one single tuple, the perceived transition of a tuple centre state can be made as complex as needed
  - this makes it possible to decouple the process's view of the tuple centre (perceived as a standard tuple space) from the actual state of a tuple centre, and to relate them so as to embed the coordination laws governing the distributed system

Tuple Centres & Hybrid Coordination

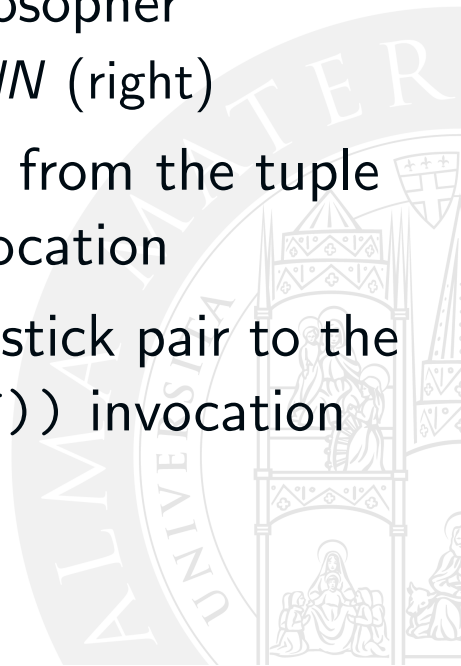
- tuple centres promote a form of hybrid coordination
  - aimed at preserving the advantages of data-driven models
  - while addressing their limitations in terms of control capabilities
- on the one hand, a tuple centre is basically an information-driven coordination medium, which is perceived as such by processes
- on the other hand, a tuple centre also features some capabilities which are typical of action-driven models, like
  - the full observability of events
  - the ability to selectively react to events
  - the ability to implement coordination rules by manipulating the interaction space



Focus on...

Dining Philosophers in ReSpecT

- the spaghetti bowl, or, more easily, the table where the bowl and the chopstick are, and the philosophers are seated, are represented by tuple centre `table`
- chopsticks are represented as tuples of the form `chop(`i`)`, where `chop(`i`)` represents the left chopstick for the ith philosopher
  - philosopher i needs chopsticks i (left) and $(i+1) mod N$ (right)
- a philosopher tries to eat by getting his chopstick pair from the tuple centre by means of a `in(chops(`i`,`i`+1 mod N))` invocation
- a philosopher starts to think by releasing his own chopstick pair to the tuple centre by means of a `out(chops(`i`,`i`+1 mod N))` invocation



Dining Philosophers in ReSpecT: Philosopher Protocol

```
philosopher(I,J) :-
    think,                          % thinking
    table ? in(chops(I,J)),         % waiting to eat
    eat,                            % eating
    table ? out(chops(I,J)), % waiting to think
    !, philosopher(I,J).
```

Dining Philosophers in ReSpecT: Philosopher Protocol

```
philosopher(I,J) :-
    think,                          % thinking
    table ? in(chops(I,J)),         % waiting to eat
    eat,                            % eating
    table ? out(chops(I,J)), % waiting to think
    !, philosopher(I,J).
```

Dining Philosophers in ReSpecT: Philosopher Protocol

```
philosopher(I,J) :-
    think,                          % thinking
    table ? in(chops(I,J)),         % waiting to eat
    eat,                            % eating
    table ? out(chops(I,J)), % waiting to think
    !, philosopher(I,J).
```

Dining Philosophers in ReSpecT: Philosopher Protocol

```
philosopher(I,J) :-
    think,                          % thinking
    table ? in(chops(I,J)),         % waiting to eat
    eat,                            % eating
    table ? out(chops(I,J)), % waiting to think
    !, philosopher(I,J).
```

Dining Philosophers in ReSpecT: Philosopher Protocol

```
philosopher(I,J) :-
    think,                          % thinking
    table ? in(chops(I,J)),         % waiting to eat
    eat,                            % eating
    table ? out(chops(I,J)), % waiting to think
    !, philosopher(I,J).
```

Dining Philosophers in ReSpecT: Philosopher Protocol

```
philosopher(I,J) :-
    think,                          % thinking
    table ? in(chops(I,J)),         % waiting to eat
    eat,                            % eating
    table ? out(chops(I,J)), % waiting to think
    !, philosopher(I,J).
```

Results

Dining Philosophers in ReSpecT: Philosopher Protocol

```
philosopher(I,J) :-
    think,                          % thinking
    table ? in(chops(I,J)),         % waiting to eat
    eat,                            % eating
    table ? out(chops(I,J)),        % waiting to think
    !, philosopher(I,J).
```

Results

+ fairness, no deadlock

Dining Philosophers in ReSpecT: Philosopher Protocol

```
philosopher(I,J) :-
    think,                          % thinking
    table ? in(chops(I,J)),         % waiting to eat
    eat,                            % eating
    table ? out(chops(I,J)),        % waiting to think
    !, philosopher(I,J).
```

Results

+ fairness, no deadlock
+ trivial philosopher's interaction protocol

Dining Philosophers in ReSpecT: Philosopher Protocol

```
philosopher(I,J) :-
    think,                          % thinking
    table ? in(chops(I,J)),         % waiting to eat
    eat,                            % eating
    table ? out(chops(I,J)),        % waiting to think
    !, philosopher(I,J).
```

Results

+ fairness, no deadlock
+ trivial philosopher's interaction protocol
? shared resources handled properly?

Dining Philosophers in ReSpecT: Philosopher Protocol

```
philosopher(I,J) :-
    think,                          % thinking
    table ? in(chops(I,J)),         % waiting to eat
    eat,                            % eating
    table ? out(chops(I,J)), % waiting to think
    !, philosopher(I,J).
```

Results

+ fairness, no deadlock
+ trivial philosopher's interaction protocol
? shared resources handled properly?
? starvation still possible?

Dining Philosophers in ReSpecT: `table` Behaviour Spec

Dining Philosophers in ReSpecT: `table` Behaviour Spec

```
reaction( out(chops(C1,C2)), (operation, completion), (    % (1)
    in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).
```

Dining Philosophers in ReSpecT: `table` Behaviour Spec

```
reaction( out(chops(C1,C2)), (operation, completion), (     % (1)
    in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (      % (2)
    out(required(C1,C2)) )).
```

Dining Philosophers in ReSpecT: `table` Behaviour Spec

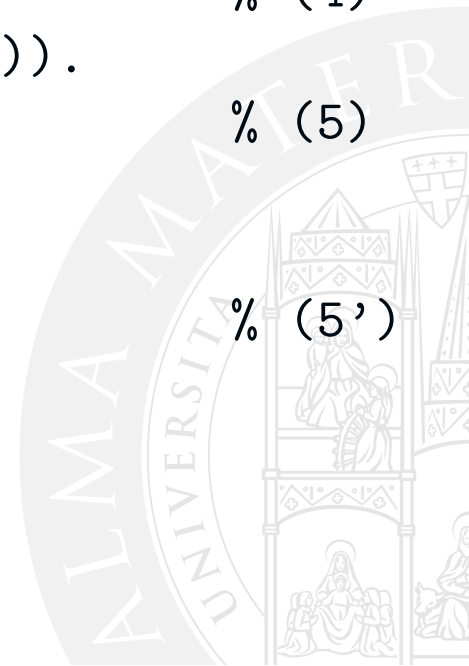
```
reaction( out(chops(C1,C2)), (operation, completion), (     % (1)
    in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (      % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (      % (3)
    in(required(C1,C2)) )).
```

Dining Philosophers in ReSpecT: `table` Behaviour Spec

```
reaction( out(chops(C1,C2)), (operation, completion), (     % (1)
    in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (      % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (      % (3)
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                 % (4)
    in(chop(C1)), in(chop(C2)),  out(chops(C1,C2)) )).
```

Dining Philosophers in ReSpecT: `table` Behaviour Spec

```
reaction( out(chops(C1,C2)), (operation, completion), (     % (1)
    in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (      % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (      % (3)
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                 % (4)
    in(chop(C1)), in(chop(C2)),  out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                         % (5)
    rd(required(C,C2)), in(chop(C)), in(chop(C2)),
    out(chops(C,C2)) )).
reaction( out(chop(C)), internal, (                         % (5')
    rd(required(C1,C)), in(chop(C1)), in(chop(C)),
    out(chops(C1,C)) )).
```



Dining Philosophers in ReSpecT: Results

Results

protocol	no deadlock
protocol	fairness
protocol	trivial philosopher's interaction protocol
tuple centre	shared resources handled properly
-	starvation still possible

Distributed Dining Philosophers

Dining Philosophers in a distributed setting

- N philosophers are distributed along the network
  - each philosopher is assigned a seat, represented by the tuple centre `seat(i,j)`
  - `seat(i,j)` denotes that the associated philosopher needs chopstick pair `chops(i,j)` so as to eat
- each chopstick `i` is represented as a tuple `chop(i)` in the `table` tuple centre
- each philosopher expresses his intention to eat / think by emitting a tuple `wanna_eat` / `wanna_think` in his `seat(i,j)` tuple centre
  - everything else is handled automatically in ReSpecT, embedded in the tuple centre behaviour
- N individual tuple centres (`seat(i,j)`) + 1 social tuple centre (`table`) connected in a star network

Distributed Dining Philosophers: Individual Interaction

Philosopher–seat interaction (*use*)

- four states, represented by tuple `philosopher(_)`
  - `thinking`, `waiting_to_eat`, `eating`, `waiting_to_think`
- determined by
  - the `out(wanna_eat)` / `out(wanna_think)` invocations, expressing the philosopher's intentions
  - the interaction with the `table` tuple centre, expressing the availability of `chop` resources
- tuple `chops(i,j)` only occurs in tuple centre `seat(i,j)` in the `philosopher(eating)` state
- 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`

ReSpecT code for seat(*i*, *j*) tuple centres

```
reaction( out(wanna_eat), (operation, invocation), (                    % (1)
    in(philosopher(thinking)), out(philosopher(waiting_to_eat)),
    current_target(seat(C1,C2)), table@node ? in(chops(C1,C2)) )).
reaction( out(wanna_eat), (operation, completion),                      % (2)
    in(wanna_eat)).
reaction( in(chops(C1,C2)), (link_out, completion),                     % (3)
    in(philosopher(waiting_to_eat)), out(philosopher(eating)),
    out(chops(C1,C2)) )).
reaction( out(wanna_think), (operation, invocation), (                  % (4)
    in(philosopher(eating)), out(philosopher(waiting_to_think)),
    current_target(seat(C1,C2)), in(chops(C1,C2)),
    table@node ? out(chops(C1,C2)) )).
reaction( out(wanna_think), (operation, completion),                    % (5)
    in(wanna_think) ).
reaction( out(chops(C1,C2)), (link_out, completion), (                  % (6)
    in(philosopher(waiting_to_think)), out(philosopher(thinking)) )).
```

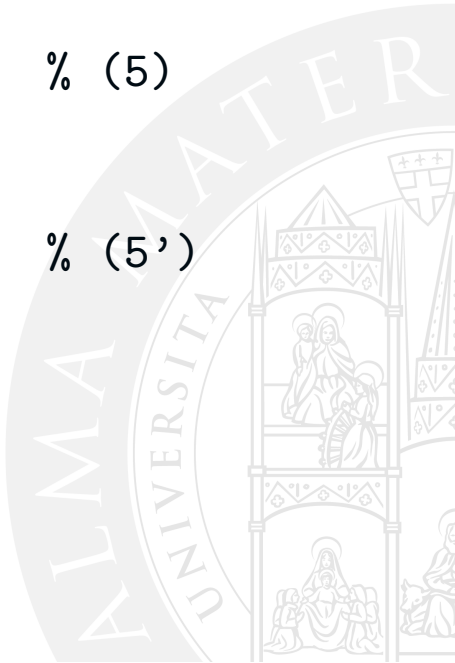
Distributed Dining Philosophers: Social Interaction

Seat–table interaction (*link*)

- tuple centre `seat(i,j)` requires / returns tuple `chops(i,j)` from / to `table` tuple centre
- tuple centre `table` transforms tuple `chops(i,j)` into a tuple pair `chop(i)`, `chop(j)` whenever required, and back `chop(i)`, `chop(j)` into `chops(i,j)` whenever required and possible

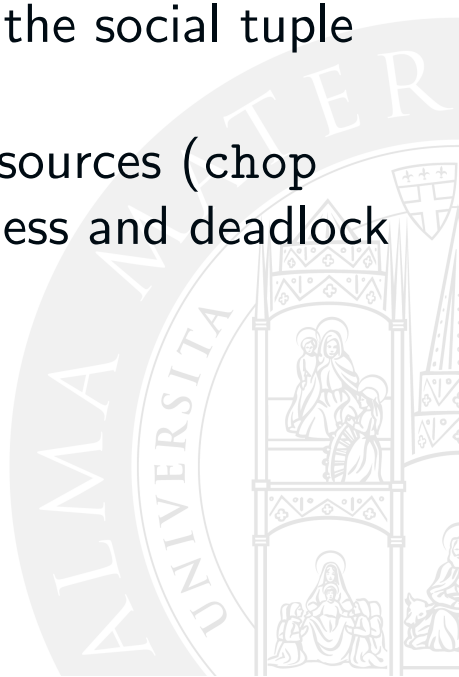
ReSpecT code for table tuple centre

```
reaction( out(chops(C1,C2)), (link_in, completion), (      % (1)
    in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (link_in, invocation), (       % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (link_in, completion), (       % (3)
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                % (4)
    in(chop(C1)), in(chop(C2)), out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                        % (5)
    rd(required(C,C2)), in(chop(C)), in(chop(C2)),
    out(chops(C,C2)) )).
reaction( out(chop(C)), internal, (                        % (5')
    rd(required(C1,C)), in(chop(C1)), in(chop(C)),
    out(chops(C1,C)) )).
```



Distributed Dining Philosophers: Features I

- full separation of concerns
  - philosophers just express their intentions, in terms of simple tuples
  - individual tuple centres – `seat(i,j)` – handle individual behaviours and state, and mediate interaction of individuals with the social tuple centre—`table`
  - the social tuple centre – `table` – deals with shared resources (`chop` tuples) and ensures global system properties, like fairness and deadlock avoidance



Distributed Dining Philosophers: Features II

At any time, one could look at the coordination media, and find exactly the consistent representation of the current distributed state

- properly distributed, suitably encapsulated
  - the state of shared resources is in the shared distributed abstraction, the state of single processes is into individual local abstraction
- accessible, represented in a declarative way
  - the state of individual philosophers is exposed through accessible media as far as the portion representing their social interaction is concerned

Focus on...

Reaction Specification Tuples (ReSpecT)

As a *behaviour specification language*, ReSpecT

- enables the definition of computations within a tuple centre—called reactions
- makes it possible to associate such reactions to events occurring in a tuple centre

ReSpecT twofold interpretation

So, ReSpecT has both a declarative and a procedural part

Declarative ReSpecT

As a specification language, ReSpecT allows *events* to be declaratively associated to *reactions* by means of specific logic tuples, called specification tuples, whose form is `reaction(`E`,`G`,`R`)` [Omicini, 2007].

ReSpecT specification tuple

In short, given a ReSpecT event Ev, a specification tuple `reaction(`E`,`G`,`R`)` associates a reaction $R\theta$ to Ev if and only if $\theta = mgu(E,Ev)$ – where *mgu* is the most general unifier, as defined in logic programming – *and* guard predicate `G` is true

ReSpecT Syntax

Currently, TuCSoN supports the following ReSpecT syntax

- `E` (Event) — any TuCSoN primitive (except for `get_s` and `set_s`)
- `G` (Guard) — any combination of
  - `invocation` : `true` $\iff$ ReSpecT VM is in the invocation phase
  - `completion` : `true` $\iff$ ReSpecT VM is in the completion phase
  - `success` : `true` $\iff$ ReSpecT primitive succeeded
  - `failure` : `true` $\iff$ ReSpecT primitive failed
  - `endo` : `true` $\iff$ the cause of the event is the current tuple centre (tc)
  - `exo` : `true` $\iff$ the *cause* of the event *is not* the current tc
  - `intra` : `true` $\iff$ the target of the operation is the current tc
  - `inter` : `true` $\iff$ the target of the operation *is not* the current tc
  - `from_agent` : `true` $\iff$ the source of the ReSpecT event is an agent
  - `from_tc` : `true` $\iff$ the *source* of the ReSpecT event is a tuple centre
- `R` (Reaction) — any TuCSoN primitive (except for `get_s` and `set_s`) | any Prolog computation | any "mix" of the two

For the full formal syntax, please refer to [Omicini, 2007], Table 1

Knowledge in ReSpecT Tuple Centres

By adopting the declarative interpretation, a ReSpecT tuple centre has then a *twofold nature*: a theory of communication – the set of the ordinary tuples – and a theory of coordination—the set of the specification tuples.

Support for reasoning

This allows in principle intelligent agents to reason about the state of *collaboration activities* and to possibly affect their dynamics

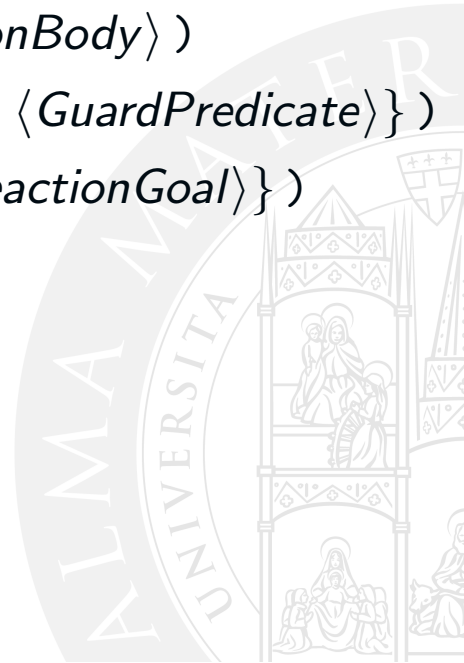
ReSpecT Basic Syntax for Reactions

Logic tuples

- ReSpecT tuple centres adopt logic tuples for both ordinary tuples and specification tuples
- ordinary tuples are simple first-order logic (FOL) facts, written with a Prolog syntax
  - while ordinary logic tuples are typically ground facts, there is nothing to constrain them to be such
- specification tuples are logic tuples of the form `reaction(`E`,`G`,`R`)`
  - if event Ev occurs in the tuple centre,
  - which matches event descriptor E such that $\theta = mgu(E,Ev)$, and
  - guard G is true,
  - then reaction $R\theta$ to Ev is triggered for execution in the tuple centre

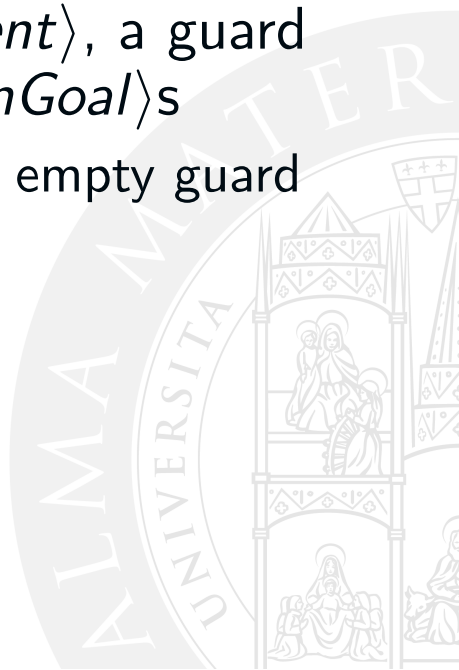
ReSpecT Syntax: Structure

$$
\begin{aligned}
\langle \mathit{Specification} \rangle &::= \{ \langle \mathit{SpecificationTuple} \rangle \ . \} \\
\langle \mathit{SpecificationTuple} \rangle &::= \texttt{reaction(} \langle \mathit{Event} \rangle \texttt{ ,} [\langle \mathit{Guard} \rangle \texttt{ ,}] \langle \mathit{ReactionBody} \rangle \texttt{)} \\
\langle \mathit{Guard} \rangle &::= \langle \mathit{GuardPredicate} \rangle \mid \texttt{(} \langle \mathit{GuardPredicate} \rangle \{ \texttt{ ,} \langle \mathit{GuardPredicate} \rangle \} \texttt{)} \\
\langle \mathit{ReactionBody} \rangle &::= \langle \mathit{ReactionGoal} \rangle \mid \texttt{(} \langle \mathit{ReactionGoal} \rangle \{ \texttt{ ,} \langle \mathit{ReactionGoal} \rangle \} \texttt{)}
\end{aligned}
$$



ReSpecT Behaviour Specification

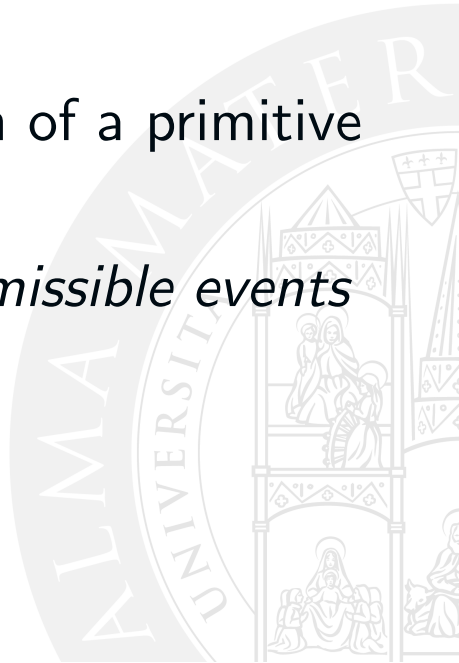
- a behaviour specification $\langle Specification \rangle$ is a logic theory of FOL tuples `reaction/3`
- a specification tuple contains an event descriptor $\langle Event \rangle$, a guard $\langle Guard \rangle$, and a sequence $\langle ReactionBody \rangle$ of $\langle ReactionGoal \rangle$s
  - a `reaction/2` specification tuple implicitly defines an empty guard



ReSpecT Event Descriptor

$$\langle Event \rangle \quad ::= \quad \langle Predicate \rangle \, (\, \langle Tuple \rangle \,) \mid \ldots$$

- the simplest event descriptor $\langle Event \rangle$ is the invocation of a primitive $\langle Predicate \rangle \, (\, \langle Tuple \rangle \,)$
- an event descriptor $\langle Event \rangle$ is used to match with *admissible events*



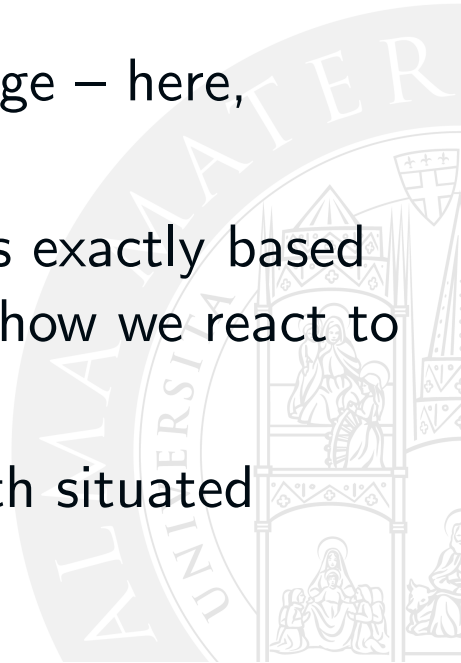
ReSpecT Admissible (Tuple Centre) Event

$$
\begin{array}{rcl}
\langle \mathit{TCEvent} \rangle & ::= & \langle \mathit{OpEvent} \rangle \mid \ldots \\
\langle \mathit{OpEvent} \rangle & ::= & \langle \mathit{OpStartCause} \rangle \, , \langle \mathit{OpEventCause} \rangle \, , \langle \mathit{OpResult} \rangle \\
\langle \mathit{OpStartCause} \rangle & ::= & \langle \mathit{CoordOp} \rangle \, , \langle \mathit{AgentId} \rangle \, , \langle \mathit{TCId} \rangle \\
\langle \mathit{OpEventCause} \rangle & ::= & \langle \mathit{OpStartCause} \rangle \mid \langle \mathit{LinkOp} \rangle \, , \langle \mathit{TCId} \rangle \, , \langle \mathit{TCId} \rangle \\
\langle \mathit{OpResult} \rangle & ::= & \langle \mathit{Tuple} \rangle \, , \ldots
\end{array}
$$

- a ReSpecT admissible event includes its *prime cause* $\langle \mathit{StartCause} \rangle$, its *direct cause* $\langle \mathit{EventCause} \rangle$, and the $\langle \mathit{Result} \rangle$ of the tuple centre activity
  - prime and direct cause may coincide—such as when a process invocation reaches its target tuple centre
  - or, they might be different—such as when a link primitive is invoked by a tuple centre reacting to a process' primitive invocation upon another tuple centre
  - the result is undefined in the invocation stage: it is defined in the completion stage
- a reaction specification tuple `reaction(`E`,`G`,`R`)` and an admissible event ϵ match if E unifies with the $\langle \mathit{CoordOp} \rangle \mid \langle \mathit{LinkOp} \rangle$ part of $\epsilon.\langle \mathit{OpEventCause} \rangle$

Event Model vs. Event Representation

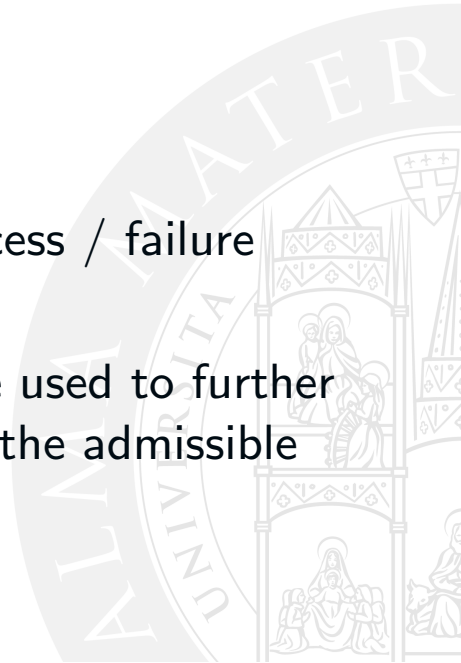
- understanding the difference between ReSpecT admissible events $\langle TCEvent \rangle$ and event descriptors $\langle Event \rangle$ is essential not to understand ReSpecT – who cares, after all – but first of all to understand the main issues of pervasive systems
- admissible events is how we capture and model all the relevant events: essentially, our *ontology for events*
- event descriptors is how we write events in our language – here, ReSpecT –: essentially, our *language for events*
- the ReSpecT VM is where the two things clash, and is exactly based on that: it's how we capture and observe events, and how we react to them properly
- this is an essential point in *any* technology dealing with situated computations



ReSpecT Guards

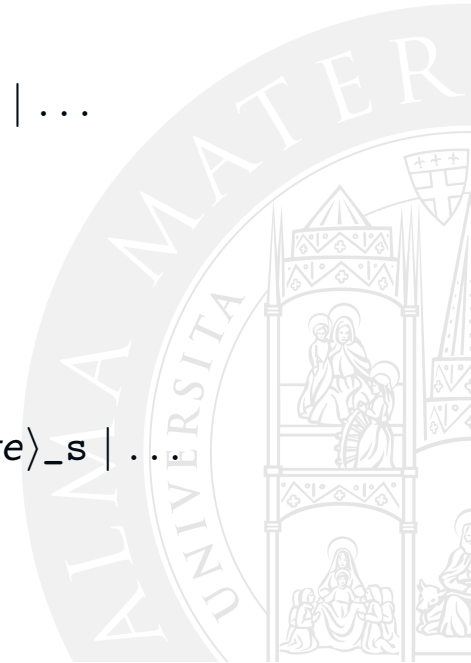
$$
\begin{array}{rcl}
\langle \mathit{Guard} \rangle & ::= & \langle \mathit{GuardPredicate} \rangle \mid \\
 & & (\ \langle \mathit{GuardPredicate} \rangle \ \{ \texttt{,} \ \langle \mathit{GuardPredicate} \rangle \}\ ) \\
\langle \mathit{GuardPredicate} \rangle & ::= & \texttt{request} \mid \texttt{response} \mid \texttt{success} \mid \texttt{failure} \\
 & & \texttt{endo} \mid \texttt{exo} \mid \texttt{intra} \mid \texttt{inter} \\
 & & \texttt{from\_agent} \mid \texttt{to\_agent} \mid \texttt{from\_tc} \mid \texttt{to\_tc} \mid \ldots
\end{array}
$$

- a triggered reaction is actually executed only if its guard is true
- all guard predicates are ground ones, so they have always a success / failure semantics
- guard predicates concern properties of the event, so they can be used to further select some classes of events after the initial matching between the admissible event and the event descriptor



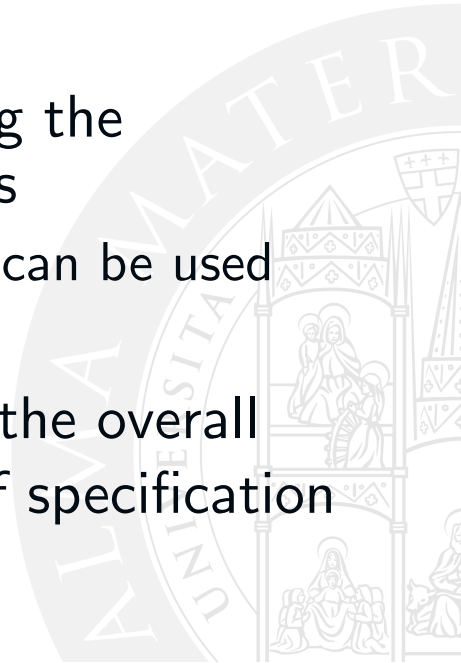
ReSpecT Reactions I

$$
\begin{array}{rcl}
\langle \textit{ReactionGoal} \rangle & ::= & \langle \textit{Predicate} \rangle \texttt{(} \langle \textit{Tuple} \rangle \texttt{)} \mid \\
& & \langle \textit{TupleCentre} \rangle \texttt{?} \langle \textit{Predicate} \rangle \texttt{(} \langle \textit{Tuple} \rangle \texttt{)} \mid \\
& & \langle \textit{ObservationPredicate} \rangle \texttt{(} \langle \textit{Tuple} \rangle \texttt{)} \mid \\
& & \langle \textit{ComputationGoal} \rangle \mid \texttt{(} \langle \textit{ReactionGoal} \rangle \texttt{;} \langle \textit{ReactionGoal} \rangle \texttt{)} \mid \\
& & \ldots \\
\langle \textit{Predicate} \rangle & ::= & \langle \textit{StatePredicate} \rangle \mid \langle \textit{ForgePredicate} \rangle \\
\langle \textit{StatePredicate} \rangle & ::= & \langle \textit{BasicPredicate} \rangle \mid \langle \textit{PredicativePredicate} \rangle \mid \ldots \\
\langle \textit{BasicPredicate} \rangle & ::= & \langle \textit{GetterPredicate} \rangle \mid \langle \textit{SetterPredicate} \rangle \\
\langle \textit{GetterPredicate} \rangle & ::= & \texttt{in} \mid \texttt{rd} \mid \texttt{no} \\
\langle \textit{SetterPredicate} \rangle & ::= & \texttt{out} \\
\langle \textit{PredicativePredicate} \rangle & ::= & \langle \textit{GetterPredicate} \rangle \texttt{p} \\
\langle \textit{ForgePredicate} \rangle & ::= & \langle \textit{BasicPredicate} \rangle \texttt{\_s} \mid \langle \textit{PredicativePredicate} \rangle \texttt{\_s} \mid \ldots
\end{array}
$$



ReSpecT Reactions II

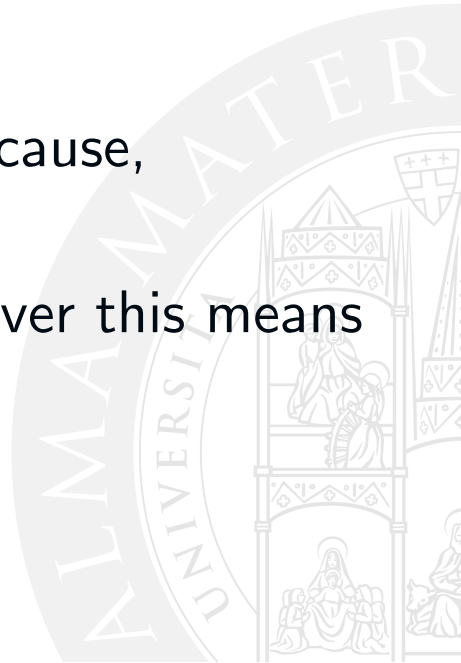
- a reaction goal is either a primitive invocation (possibly, a link), a predicate recovering properties of the event, or some logic-based computation
- sequences of reaction goals are executed transactionally with an overall success / failure semantics
- tuple centre predicates are uniformly used for agent invocations, internal operations, and link invocations
- the same predicates are substantially used for changing the specification state, with essentially the same semantics
  - *pred*_s invocations affect the specification state, and can be used within reactions, also as links
- `no` works as a test for absence, `get` and `set` work on the overall theory (either the one of ordinary tuples, or the one of specification tuples)



ReSpecT Observation Predicates

$$\begin{aligned} \langle \textit{ObservationPredicate} \rangle &::= \langle \textit{EventView} \rangle \_ \langle \textit{EventInformation} \rangle \\ \langle \textit{EventView} \rangle &::= \texttt{current} \mid \texttt{event} \mid \texttt{start} \\ \langle \textit{EventInformation} \rangle &::= \texttt{predicate} \mid \texttt{tuple} \mid \texttt{source} \mid \texttt{target} \mid \ldots \end{aligned}$$

- `event` & `start` clearly refer to immediate and prime cause, respectively
- `current` refers to what is currently happening, whenever this means something useful—typically, to the result



Properties of ReSpecT Tuple Centres

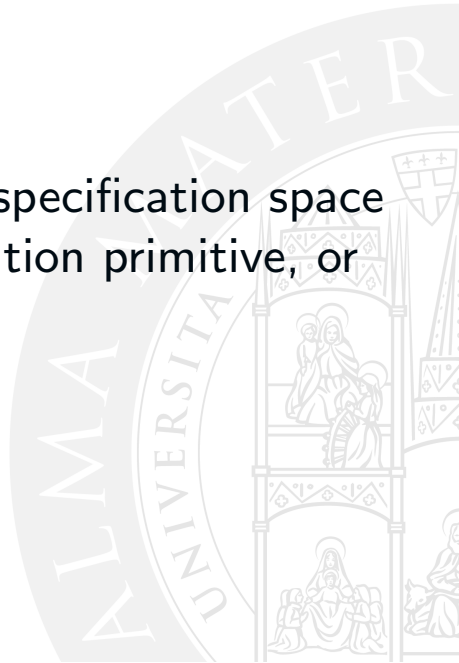
- ReSpecT tuple centres
  - encapsulate knowledge in terms of logic tuples
  - encapsulates behaviour in terms of ReSpecT specifications
- ReSpecT tuple centres are
  - inspectable
  - malleable
  - linkable

Inspectability of ReSpecT Tuple Centres

- ReSpecT tuple centres: twofold space for tuples
  - tuple space: ordinary (logic) tuples
    - for knowledge, information, messages, communication
    - working as the (logic) *theory of communication* for distributed systems
  - specification space: specification (logic, ReSpecT) tuples
    - for behaviour, function, coordination
    - working as the (logic) *theory of coordination* for distributed systems
- both spaces are inspectable
  - by engineers, via ReSpecT inspectors
  - by processes, via `rd` & `no` primitives
    - `rd` & `no` for the tuple space; `rd_s` & `no_s` for the specification space
    - either directly or indirectly, through either a coordination primitive, or another tuple centre

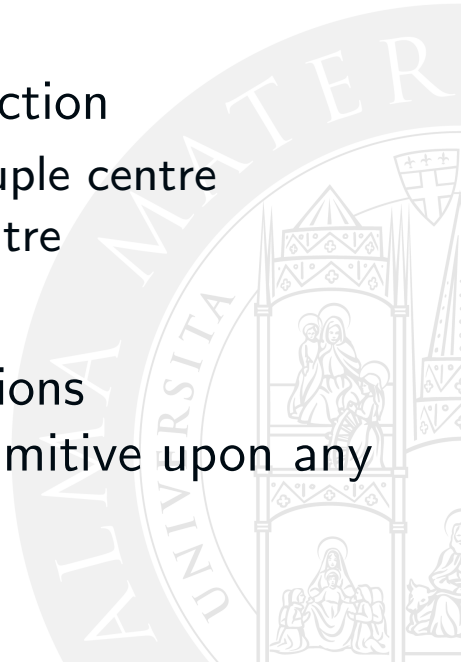
Malleability of ReSpecT Tuple Centres

- the behaviour of a ReSpecT tuple centre is defined by the ReSpecT tuples in the specification space
  - it can be adapted / changed by changing its ReSpecT specification
- ReSpecT tuple centres are malleable
  - by engineers, via ReSpecT tools
  - by processes, via `in` & `out` primitives
    - `in` & `out` for the tuple space; `in_s` & `out_s` for the specification space
    - either directly or indirectly, through either a coordination primitive, or another tuple centre



Linkability of ReSpecT Tuple Centres

- every tuple centre coordination primitive is also a ReSpecT primitive for reaction goals, and a primitive for linking, too
  - all primitives are asynchronous
    - so they do not affect the transactional semantics of reactions
  - all primitives have a request / response semantics
    - including `out` / `out_s`
    - so reactions can be defined to handle both primitive invocations & completions
  - all primitives could be executed within a ReSpecT reaction
    - as either a reaction goal executed within the same tuple centre
    - or as a link primitive invoked upon another tuple centre
- ReSpecT tuple centres are linkable
  - by using tuple centre identifiers within ReSpecT reactions
  - any ReSpecT reaction can invoke any coordination primitive upon any tuple centre in the network



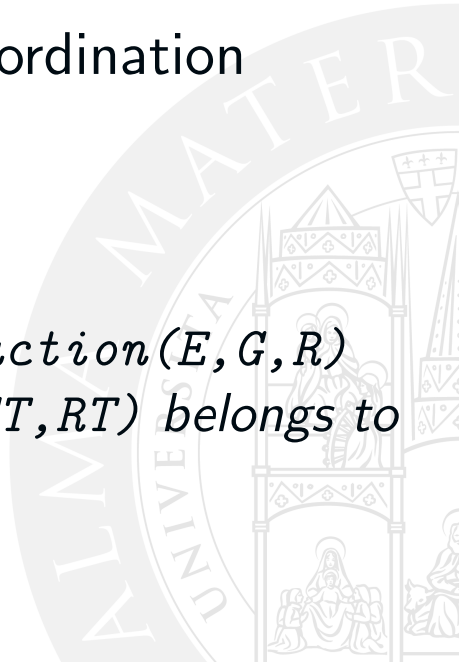
Next in Line...

Focus on...

Meta-Coordination Language

- the TuCSoN meta-coordination language allows agents to program ReSpecT tuple centres by executing *meta-coordination operations*
- TuCSoN provides coordinables with meta-coordination primitives, allowing agents to read, write, consume ReSpecT specification tuples in tuple centres and also to synchronise on them
- meta-coordination operations are built out of meta-coordination primitives and of the ReSpecT specification languages
  - the *specification language*
  - the *specification template language*

! *in the following, whenever unspecified, we assume that* `reaction(E,G,R)` *belongs to the specification language, and* `reaction(ET,GT,RT)` *belongs to the specification template language*



Specification & Specification Template Languages

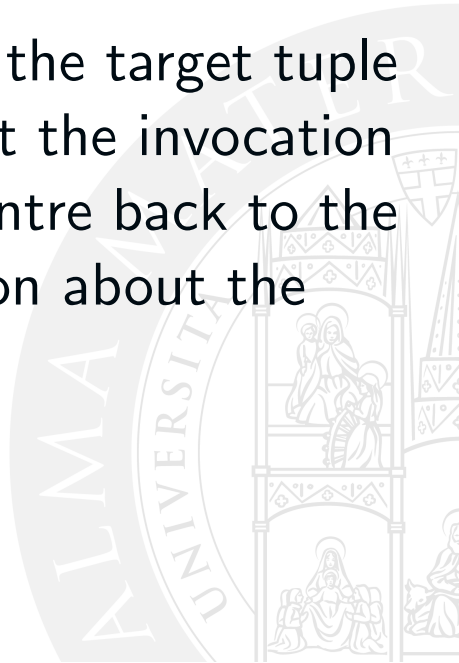
- given that the TuCSoN coordination medium is the *logic-based* ReSpecT tuple centre, both the specification and the specification template languages are logic-based too—and defined by ReSpecT
- more precisely
  - any ReSpecT reaction is an admissible TuCSoN specification tuple...
  - ... and an admissible TuCSoN specification template

Meta-Coordination Operations

- any TuCSoN *meta-coordination operation* is invoked by a source agent on a target tuple centre, which is in charge of its execution
- in the same way as TuCSoN coordination operations, any meta-coordination operation has two phases

  invocation — the *request* from the source agent to the target tuple centre, carrying all the information about the invocation

  completion — the *response* from the target tuple centre back to the source agent, including all the information about the operation execution by the tuple centre



Abstract Syntax

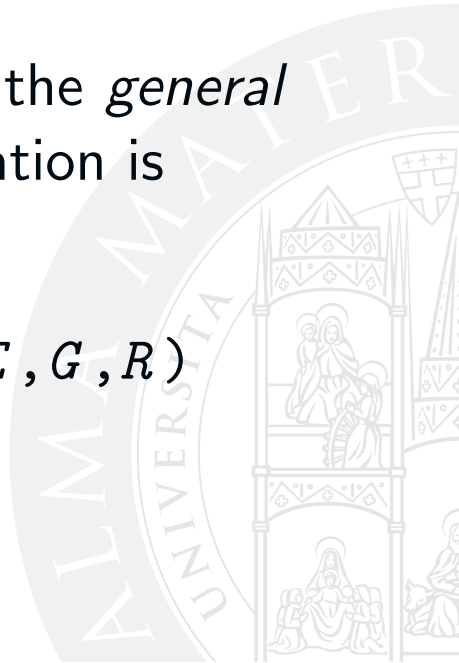
- the abstract syntax of a meta-coordination operation *op*_s invoked on a target tuple centre *tcid* is

  tcid ? *op*_s

  where tcid is the tuple centre *full name*
- given the structure of the full name of a tuple centre, the *general abstract syntax* of a TuCSoN meta-coordination operation is

  tname @ *netid* : *portno* ? *op*_s

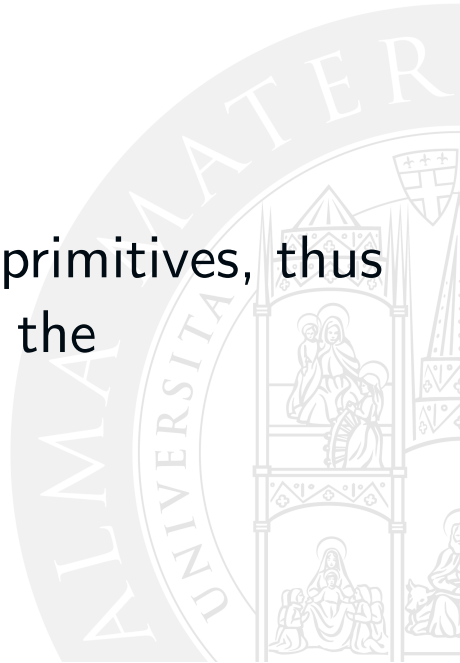
  default @ localhost : 20504 ? out_s(*E*,*G*,*R*)



Focus on...

Meta-Coordination Primitives

- the TuCSoN meta-coordination language provides the following meta-coordination primitives to build meta-coordination operations
  - `out_s`
  - `rd_s`, `rdp_s`
  - `in_s`, `inp_s`
  - `no_s`, `nop_s`
  - `get_s`
  - `set_s`
- as you can see, meta-primitives perfectly match basic primitives, thus allowing a uniform access to both the tuple space and the specification space in a TuCSoN tuple centre



Next in Line...

Focus on...

Meta-Coordination Experiments (1/2) I

Let's try! (1)

1. launch a TuCSoN node on default port

```
java -cp libs/tucson.jar:libs/2p.jar alice.tucson.service.TucsonNodeService
```

2. launch the CLI

```
java -cp libs/tucson.jar:libs/2p.jar
alice.tucson.service.tools.CommandLineInterpreter
```

3. launch TuCSoN Inspector tool — that is, class `InspectorGUI` in package `alice.tucson.introspection.tools`

```
java -cp libs/tucson.jar:libs/2p.jar
alice.tucson.introspection.tools.InspectorGUI
```

Meta-Coordination Experiments (1/2) II

Let's try! (2)

4. experiment with TuCSoN meta-coordination primitives
   - add a ReSpecT reaction to a tuple centre and try to trigger it—e.g., `out_s(out(t(X)), (operation,invocation), (no(t(X)), out(first(t(X)))))`
   - add two ReSpecT reactions and see how they "chain" together
   - play with guards
   - play with *distributed reactions*
5. launch class `BagOfTaskTest` in package `ds.lab.tucson.respect.bagOfTask`

   ```
   java -cp libs/tucson.jar:libs/2p.jar:bin
   ds.lab.tucson.respect.bagOfTask.BagOfTaskTest
   ```
6. do whatever you want, given that ReSpecT is Turing-complete

   [Denti et al., 1998]

Focus on...

Procedural ReSpecT

As a reaction language, ReSpecT enables *reactions* to be procedurally defined in terms of sequences of logic reaction goals, each one either succeeding or failing.

Reactions semantics

- a ReSpecT reaction *as a whole* succeeds if and only if all its reaction goals succeed, and fails otherwise
- each reaction is executed sequentially with a transactional semantics: hence, a failed reaction has *no effect* on the state of a ReSpecT tuple centre
- all the reactions triggered by a ReSpecTevent are executed *before* serving any other event: thus, agents are transparent to any reactions chain and perceive them as atomic—along with the (meta-)coordination primitive invoked.

ReSpecT Main Execution Cycle I

Whenever the invocation of a primitive by either an agent or a tuple centre is performed

Invocation

1. an (admissible) ReSpecT event is generated and...
2. ...reaches its (the primitive) target tuple centre
3. where it is *orderly* inserted in a sort of *input queue* (*InQ*)

ReSpecT Main Execution Cycle II

When the tuple centre is idle (that is, no reaction is currently being executed)

Triggering

1. the first event ϵ in *InQ* (according to a FIFO policy) is moved to the multiset *Op* of the requests to be served
2. consequently, reactions to the invocation phase of ϵ are *triggered* by adding them to the multiset *Re* of the triggered reactions waiting to be executed
3. all triggered reactions in *Re* are then executed in a *non-deterministic order*—sequential, transactional semantics

ReSpecT Main Execution Cycle III

Chaining & linking

Each reaction may trigger

- further reactions, again to be added to *Re*
- output events representing link invocations: such events are
  1. added to the multiset *Out* of the *outgoing events*
  2. then moved to the tuple-centre *outgoing queue* *OutQ* at the end of the reaction execution—if and only if successful

ReSpecT Main Execution Cycle IV

Completion

Only when *Re* is finally empty

1. requests waiting to be served in *Op* are possibly executed by the tuple centre
2. operation/link completions are sent back to invokers

This may give raise to further reactions, associated to the completion phase of the original invocation, and executed with the same semantics specified above for the invocation phase.

Focus on...

Meta-Coordination Experiments (2/2) I

Let's try! (1)

1. launch a TuCSoN Node, Inspector, and CLI

```
java -cp libs/tucson.jar:libs/2p.jar alice.tucson.service.TucsonNodeService
java -cp libs/tucson.jar:libs/2p.jar
alice.tucson.service.tools.CommandLineInterpreter
java -cp libs/tucson.jar:libs/2p.jar
alice.tucson.introspection.tools.InspectorGUI
```

Meta-Coordination Experiments (2/2) II

Let's try! (2)

2. try to detect & follow the ReSpecT VM main cycle by analyzing TuCSoN Node outputs on the console
   - detect the invocation phase and intercept it with a ReSpecT reaction—e.g.,
     ```
     out_s( in(t(X)), (operation,invocation), ( no(t(X)), out(t(X)) ) )
     ```
   - do the same with the completion phase—e.g.,
     ```
     out_s( rd(t(X)), (operation,completion), in(t(X)) )
     ```
   - detect the triggering phase and experiment success/failure of guard predicates evaluation
   - detect the reaction execution phase and experiment their success/failure
   - try to generate linking operations and detect them, possibly intercepting them in with a ReSpecT reaction

Focus on...

Usage Example: Dining Philos I

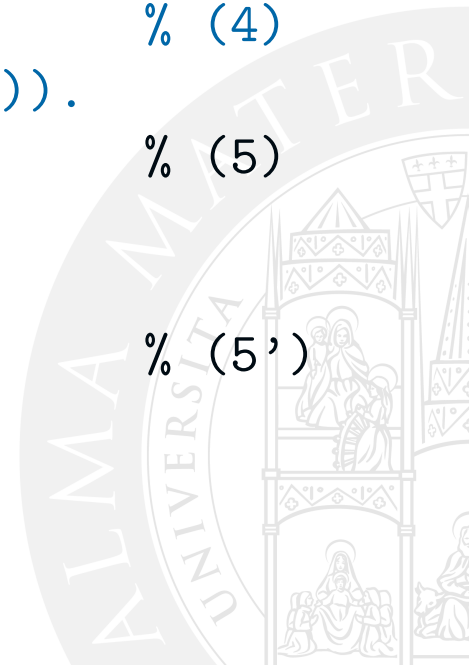
```
philosopher(I,J) :-
    think,                          % thinking
    table ? in(chops(I,J)),         % waiting to eat
    eat,                            % eating
    table ? out(chops(I,J)),        % waiting to think
!,  philosopher(I,J).
```

Results

+ fairness, no deadlock
+ trivial philosopher's interaction protocol
? shared resources handled properly?
? starvation still possible?

Usage Example: Dining Philos II

```
reaction( out(chops(C1,C2)), (operation, completion), ( % (1)
    in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (  % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (  % (3)
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (             % (4)
    in(chop(C1)), in(chop(C2)),  out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                     % (5)
    rd(required(C,C2)), in(chop(C)), in(chop(C2)),
    out(chops(C,C2)) )).
reaction( out(chop(C)), internal, (                     % (5')
    rd(required(C1,C)), in(chop(C1)), in(chop(C)),
    out(chops(C1,C)) )).
```



Usage Example: Dining Philos III

Results

protocol	no deadlock
protocol	fairness
protocol	trivial philosopher's interaction protocol
tuple centre	shared resources handled properly
!	starvation still possible

Keep starvation in mind...

...the issue will be fixed later on using ReSpecT

ReSpecT API I

Full ReSpecT API

Class `Respect2PLibrary` in package `alice.respect.api`implements all predicates available within ReSpecT reactions, including observation predicates, and also guards

ReSpecT API II

Let's try!

Check out examples in package `ds.lab.tucson.respect.*`

1. `.bagOfTask`

   ```
   java -cp libs/tucson.jar:libs/2p.jar:bin
   ds.lab.tucson.respect.bagOfTask.BagOfTaskTest
   ```

2. `.diningPhilos`

   ```
   java -cp libs/tucson.jar:libs/2p.jar:bin
   ds.lab.tucson.respect.diningPhilosophers.DiningPhilosophersTest
   ```

3. `.distributedDiningPhilos`—linking primitives here, two nodes (e.g., 20504, 20505)

   ```
   java -cp libs/tucson.jar:libs/2p.jar:bin
   ds.lab.tucson.respect.distributedDiningPhilosophers.DDiningPhilosophersTest
   ```

Focus on...

External APIs

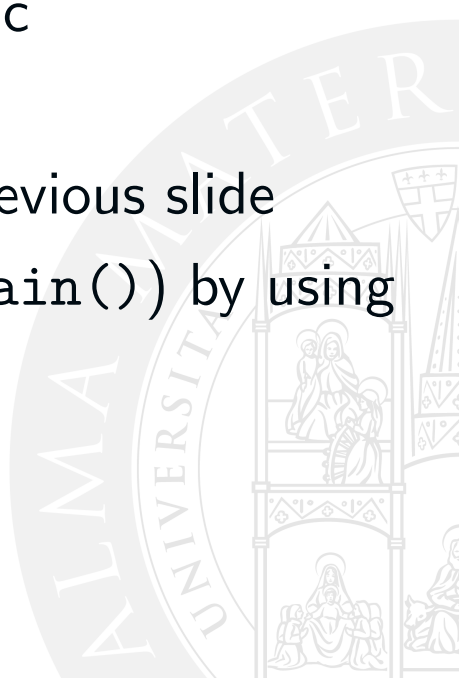
Uniform w.r.t. TuCSoN APIs accessing the ordinary tuples space:

1. build a `TucsonAgentId`
2. get a TuCSoN ACC allowing ReSpecT programming (e.g. `SynchACC` extends `SpecificationSynchACC`)
3. define the tuplecentre target of your meta-coordination operations
4. build a specification tuple for each construct of a ReSpecT reaction
   - → `LogicTuple event = LogicTuple.parse("out(t(X))");`
   - → `LogicTuple guards = LogicTuple.parse("(completion, success)");`
   - → `LogicTuple body = LogicTuple.parse("(out(in(t(X))), out(tuple(X)))");`
5. perform the meta-coordination operation using a meta-coordination primitive
   - → `ITucsonOperation op = acc.out_s(tid, event, guards, body, null);`
6. check requested operation success
7. get requested operation result

Extension APIs

Again, nothing new should be done except exploiting a suitable ACC:

1. extend `alice.tucson.api.TucsonAgent` base class
2. choose one of the given constructors, e.g.
3. override `main()` method with your agent business logic
4. get your ACC from the super-class
5. do what you want to do following steps 3 – 7 from previous slide
6. instantiate your agent and start its execution cycle (`main()`) by using method `go()`



Next in Line...

Situatedness in the Spatio-Temporal Fabric I

What is Situatedness?

- situatedness is in short the property of systems of being *immersed* in their environment
- that is, of being capable to *perceive* and *produce* environment change, by suitably dealing with environment events
- mobile, adaptive, and pervasive computing systems have emphasised the key role of situatedness for nowadays computational systems

  [Zambonelli et al., 2011]

Situatedness in the Spatio-Temporal Fabric II

Situatedness & context-awareness

- computational systems are more and more affected by their physical nature
- this is not due to a lack of abstraction: indeed, we are suitably layering systems – including hardware and software layers –, where separation between the different layers is clear and well defined
- instead, this is mostly due to the increasingly complex requirements for our computational systems, which mandate for an ever-increasing *context-awareness* [Baldauf et al., 2007]
- defining the notion of *context* is a complex matter [Omicini, 2002], with its boundaries typically blurred with the notion of *environment*, in particular in the field of multi-agent systems [Weyns et al., 2007]

Situatedness in the Spatio-Temporal Fabric III

Context-awareness: space & time

- mobile computing scenarios have made clear that *situatedness* of computational systems nowadays requires at least *awareness of the spatio-temporal fabric*
- that is, any non-trivial system needs to know *where* it is working, and *when*, in order to effectively perform its function
- in its most general acceptation, then, any environment for a computational system is first of all made of space and time

Space & time vs. coordination

- why, and to which extent is this a *coordination issue*?
- why, and to which extent is this a *tuple-based coordination issue*?

Focus on...

Dining Philosophers in ReSpecT: Starvation?

What is the problem?

- the problem is *time*: no one keeps track of time here, and starvation is a matter of time
- how can we handle time here?
  - is synchronisation not enough for the purpose?
- of course not: to avoid problems like starvation, we need the ability of defining *time-dependent* coordination policies

What is the solution?

- in order to define time-dependent coordination policies, a time-aware coordination medium is needed

Time-aware Coordination Media I

Requirements for a time-aware coordination media

A time-aware coordination medium should satisfy the following requirements [Omicini et al., 2007]

1. time has to be an integral part of the ontology of a coordination medium
2. (physical) time has to be explicitly embedded into the coordination medium working cycle
3. a coordination medium should allow coordination policies to talk about time
4. a coordination medium should be able to capture time events, and to react appropriately
5. a coordination medium should allow coordination policies to be changed over time

Time-aware Coordination Media II

Physical time in the medium ontology & working cycle *(reqs. 1, 2)*

1. time has to be an integral part of the ontology of a coordination medium
2. (physical) time has to be explicitly embedded into the coordination medium working cycle

- the ReSpecT admissible event model is extended to include *time*. For instance, in the case of $\langle OpEvent \rangle$

$$\begin{aligned} \langle OpStartCause \rangle &::= \langle CoordOp \rangle , \langle AgentId \rangle , \langle TCId \rangle , \langle Time \rangle \\ \langle OpEventCause \rangle &::= \langle OpStartCause \rangle \mid \\ &\quad \langle LinkOp \rangle , \langle TCId \rangle , \langle TCId \rangle , \langle Time \rangle \end{aligned}$$

- every ReSpecT VM executes on a sequential machine making an expression of physical time available

→ at every transition of the ReSpecT VM, physical time is always available to any ReSpecT computation

Time-aware Coordination Media III

Coordination policies: talking about time *(req. 3)*

3. a coordination medium should allow coordination policies to talk about time

- ReSpecT observation predicates are extended with time

$$\begin{aligned} \langle \textit{ObservationPredicate} \rangle &::= \langle \textit{EventView} \rangle \texttt{\_} \langle \textit{EventInformation} \rangle \\ \langle \textit{EventView} \rangle &::= \texttt{current} \mid \texttt{event} \mid \texttt{start} \\ \langle \textit{EventInformation} \rangle &::= \ldots \mid \texttt{time} \mid \ldots \end{aligned}$$

- two ReSpecT guard predicates are introduced

$$\langle \textit{GuardPredicate} \rangle ::= \ldots \mid \texttt{before(} \langle \textit{Time} \rangle \texttt{)} \mid \texttt{after(} \langle \textit{Time} \rangle \texttt{)} \mid \ldots$$

- along with the obvious alias

$$\texttt{between(} \langle \textit{Time} \rangle \texttt{,} \langle \textit{Time} \rangle \texttt{)}$$

Time-aware Coordination Media IV

Coordination policies: talking about time *(req. 3 – contd.)*

In the overall, ReSpecT is extended with time

- by introducing some temporal predicates and guards to get information about both tuple-centre and event time

observation `current_time(?Time)`, `start_time(?Time)`, `event_time(?Time)`

guard `before(@Time)`, `after(@Time)`, `between(@MinTime,@MaxTime)`

Time-aware Coordination Media V

Capturing time events *(req. 4)*

4. a coordination medium should be able to capture time events, and to react appropriately

- the ReSpecT *admissible (tuple centre) event* model is extended to include time events

$$
\begin{array}{rcl}
\langle \mathit{TCEvent} \rangle & ::= & \langle \mathit{OpEvent} \rangle \mid \langle \mathit{TEvent} \rangle \mid \ldots \\
\langle \mathit{TEvent} \rangle & ::= & \langle \mathit{TStartCause} \rangle \, , \langle \mathit{TEventCause} \rangle \, , \langle \mathit{TResult} \rangle \, , \langle \mathit{Time} \rangle \\
\langle \mathit{TStartCause} \rangle & ::= & \langle \mathit{TOp} \rangle \, , \texttt{time} , \langle \mathit{TCId} \rangle \\
\langle \mathit{TEventCause} \rangle & ::= & \langle \mathit{TStartCause} \rangle \\
\langle \mathit{TOp} \rangle & ::= & \texttt{time(} \langle \mathit{Time} \rangle \texttt{)} \\
\langle \mathit{TResult} \rangle & ::= & \langle \mathit{TOp} \rangle \, , \ldots
\end{array}
$$

- correspondingly, the ReSpecT *event descriptor* is extended, too

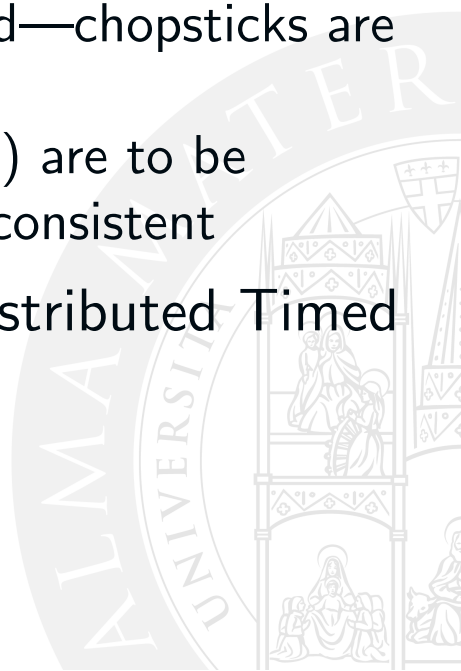
$$\langle \mathit{Event} \rangle \quad ::= \quad \langle \mathit{Predicate} \rangle \, \texttt{(} \langle \mathit{Tuple} \rangle \, \texttt{)} \mid \texttt{time(} \langle \mathit{Time} \rangle \, \texttt{)} \mid \ldots$$

making it possible to specify reactions to *time events*:

```
reaction(time(@Time), Guard, Body).
```

Timed Dining Philosophers

- an example of time-dependent coordination
- `table` tuple centre stores the maximum amount of time for any process (philosopher) to use the resource (to eat using chops)
  - in terms of a tuple `max_eating_time(@Time)`
  - if this time expires the locks are automatically released—chopsticks are re-inserted by the table tuple centre
  - late releases (by processes through `seat` tuple centres) are to be ignored—linkability used to make `seat` tuple centres consistent
- with a very simple extension using timed reactions, Distributed Timed Dining Philosophers are done [Omicini et al., 2005]



Timed Dining Philosophers: Philosopher

```
philosopher(I,J) :-
    think,                          % thinking
    table ? in(chops(I,J)),         % waiting to eat
    eat,                            % eating
    table ? out(chops(I,J)),        % waiting to think
!,  philosopher(I,J).
```

Timed Dining Philosophers: Philosopher

```
philosopher(I,J) :-
    think,                          % thinking
    table ? in(chops(I,J)),         % waiting to eat
    eat,                            % eating
    table ? out(chops(I,J)),        % waiting to think
!,  philosopher(I,J).
```

With respect to Dining Philosopher's protocol...

...this is left unchanged

Timed Dining Philosophers: `table` ReSpecT Code

```
reaction( out(chops(C1,C2)), (operation, completion), (     % (1)
    in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).
```

Timed Dining Philosophers: `table` ReSpecT Code

```
reaction( out(chops(C1,C2)), (operation, completion), (     % (1)
    in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (      % (2)
    out(required(C1,C2)) )).
```

Timed Dining Philosophers: `table` ReSpecT Code

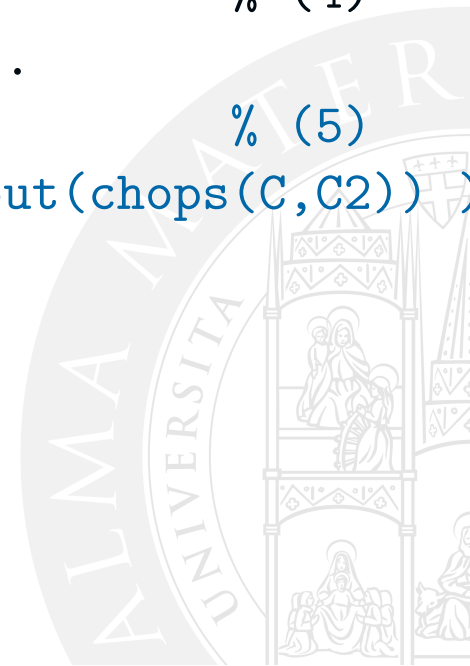
```
reaction( out(chops(C1,C2)), (operation, completion), (      % (1)
    in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (       % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (       % (3)
    in(required(C1,C2)) )).
```

Timed Dining Philosophers: `table` ReSpecT Code

```
reaction( out(chops(C1,C2)), (operation, completion), (      % (1)
    in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (       % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (       % (3)
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                  % (4)
    in(chop(C1)), in(chop(C2)), out(chops(C1,C2)) )).
```

Timed Dining Philosophers: `table` ReSpecT Code

```
reaction( out(chops(C1,C2)), (operation, completion), (      % (1)
    in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (       % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (       % (3)
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                  % (4)
    in(chop(C1)), in(chop(C2)), out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                          % (5)
    rd(required(C,C2)), in(chop(C)), in(chop(C2)), out(chops(C,C2)) )
```

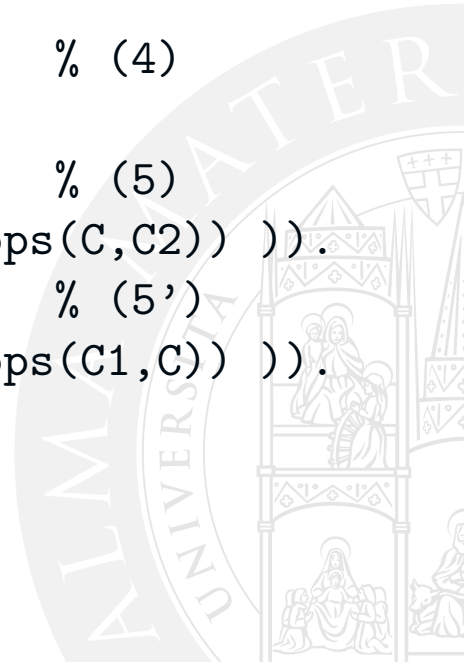


Timed Dining Philosophers: `table` ReSpecT Code

```
reaction( out(chops(C1,C2)), (operation, completion), (      % (1)
    in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (       % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (       % (3)
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                  % (4)
    in(chop(C1)), in(chop(C2)), out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                          % (5)
    rd(required(C,C2)), in(chop(C)), in(chop(C2)), out(chops(C,C2)) )
reaction( out(chop(C)), internal, (                          % (5')
    rd(required(C1,C)), in(chop(C1)), in(chop(C)), out(chops(C1,C)) )
```

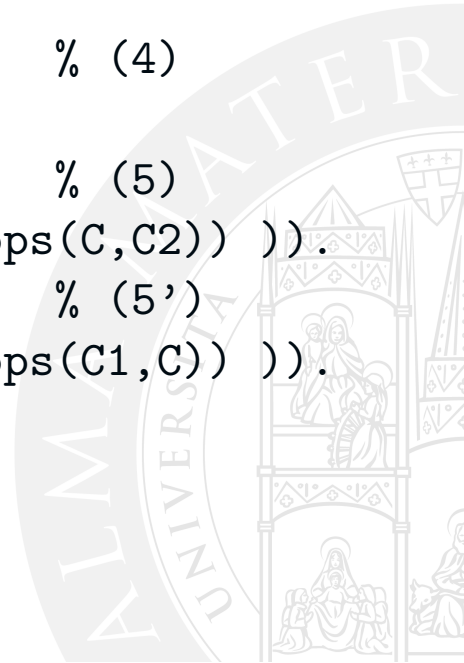
Timed Dining Philosophers: `table` ReSpecT Code

```
reaction( out(chops(C1,C2)), (operation, completion), (     % (1)
    in(chops(C1,C2)) )).
reaction( out(chops(C1,C2)), (operation, completion), (     % (1')
    out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (      % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (      % (3)
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                 % (4)
    in(chop(C1)), in(chop(C2)), out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                         % (5)
    rd(required(C,C2)), in(chop(C)), in(chop(C2)), out(chops(C,C2)) )).
reaction( out(chop(C)), internal, (                         % (5')
    rd(required(C1,C)), in(chop(C1)), in(chop(C)), out(chops(C1,C)) )).
```



Timed Dining Philosophers: `table` ReSpecT Code

```
reaction( out(chops(C1,C2)), (operation, completion), (     % (1)
    in(chops(C1,C2)) )).
reaction( out(chops(C1,C2)), (operation, completion), (     % (1')
    in(used(C1,C2,_)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (      % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (      % (3)
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                 % (4)
    in(chop(C1)), in(chop(C2)), out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                         % (5)
    rd(required(C,C2)), in(chop(C)), in(chop(C2)), out(chops(C,C2)) )).
reaction( out(chop(C)), internal, (                         % (5')
    rd(required(C1,C)), in(chop(C1)), in(chop(C)), out(chops(C1,C)) )).
```



Timed Dining Philosophers: table ReSpecT Code

```
reaction( out(chops(C1,C2)), (operation, completion), (     % (1)
    in(chops(C1,C2)) )).
reaction( out(chops(C1,C2)), (operation, completion), (     % (1')
    in(used(C1,C2,_)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (      % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (      % (3)
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                 % (4)
    in(chop(C1)), in(chop(C2)), out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                         % (5)
    rd(required(C,C2)), in(chop(C)), in(chop(C2)), out(chops(C,C2)) )).
reaction( out(chop(C)), internal, (                         % (5')
    rd(required(C1,C)), in(chop(C1)), in(chop(C)), out(chops(C1,C)) )).
reaction( in(chops(C1,C2)), (operation, completion), (      % (6)
    current_time(T), rd(max eating time(Max)), T1 is T + Max,
    out(used(C1,C2,T)),
    out_s(time(T1),(in(used(C1,C2,T)), out(chop(C1)), out(chop(C2)))) )).
```

Timed Dining Philosophers in ReSpecT: Results

Results

protocol	no deadlock
protocol	fairness
protocol	trivial philosopher's interaction protocol
tuple centre	shared resources handled properly
tuple centre	no starvation

Focus on...

What About Coordination & Space?

The issue of space awareness

- the availability of a plethora of mobile devices, in particular, is pushing forward the needs for space-awareness of computations and systems
- awareness of the *spatial context* is often essential to establish which tasks to perform, which goals to achieve, and how
- more generally, spatial issues are fundamental in many sorts of complex software systems, including intelligent, multi-agent, adaptive, and self-organising ones [Beal, 2010]
- in most of the application scenarios where situatedness plays an essential role, computation and coordination are required to be *space aware*

Situatedness & Awareness I

- spatial coordination requires
  - spatial situatedness
  - spatial awareness

  of the coordination media
- this translates in a number of *technical requirements*

Situatedness & Awareness II

Situatedness

A *space-aware coordination abstraction* should at any time be associated to an absolute positioning, both physical and virtual

In fact,

- *software abstractions* may move along a *virtual space* – typically, the network – which is usually *discrete*
- whereas *physical devices* move through a *physical space*, which is mostly *continuous*

However, software abstractions may also be hosted by mobile physical devices, thus *share* their motion.

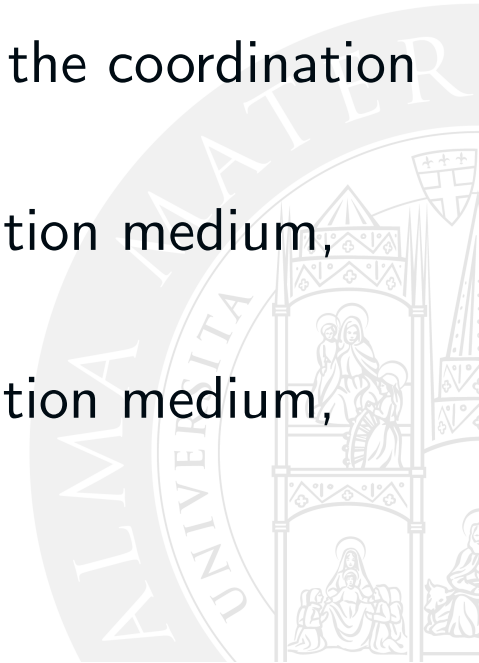
Situatedness & Awareness III

Awareness

The position of the coordination medium should be available to the *coordination laws* it contains in order to make them capable of *reasoning about space*—so, to implement space-aware coordination laws.

Also, space has to be embedded into the working cycle of the coordination medium

- a spatial event should be *generated* within a coordination medium, conceptually corresponding to *changes in space*
- then, such events should be *captured* by the coordination medium, and used to *activate* space-aware coordination laws



Space-aware Coordination Medium: Requirements I

Situatedness: requirements

1. space should intrinsically belong to the ontology of the coordination medium, in terms of *position* and *motion*—any sort of motion information, such as speed, acceleration, ...
2. both *virtual* and *physical* space acceptations should be supported
3. a notion of *locality* should be made available

Awareness: requirements

4. a coordination medium should allow coordination policies to talk about space
5. a coordination medium should be able to capture spatial events, and to react appropriately

ReSpecT Tuple Centres as Space-aware Media I

Space in medium ontology & working cycle *(reqs. 1, 2)*

1. space has to be an integral part of the ontology of a coordination medium
2. both physical & virtual position & motion have to be explicitly embedded into the coordination medium working cycle

- the ReSpecT admissible event model is extended to include *space*—physical & virtual. For instance, in the case of $\langle OpEvent \rangle$:

$$\begin{array}{rcl} \langle OpStartCause \rangle & ::= & \langle CoordOp \rangle , \langle AgentId \rangle , \langle TCId \rangle , \langle Time \rangle , \langle Place \rangle , \langle Node \rangle \\ \langle OpEventCause \rangle & ::= & \langle OpStartCause \rangle \mid \\ & & \langle LinkOp \rangle , \langle TCId \rangle , \langle TCId \rangle , \langle Time \rangle , \langle Place \rangle , \langle Node \rangle \end{array}$$

- every ReSpecT VM executes on a physical machinery and network node making an expression of physical & virtual positioning available

→ at every transition of the ReSpecT VM, physical & virtual positioning is always available to any ReSpecT computation

ReSpecT Tuple Centres as Space-aware Media II

Locality in TuCSoN *(req. 3)*

3. a notion of *locality* should be made available

   - ReSpecT tuple centres have unique identities and IDs within a TuCSoN node
   - any $\langle CoordOp \rangle$ / $\langle LinkOp \rangle$ can refer to a simple tuple centre identifier, using a TuCSoN node as a straightforward *local (coordination) space*
   - adding a node reference to a $\langle CoordOp \rangle$ / $\langle LinkOp \rangle$ shifts everything to the *global (coordination) space*
   - this however is just a notion of virtual locality

ReSpecT Tuple Centres as Space-aware Media III

Coordination policies: talking about space *(req. 4)*

4. a coordination medium should allow coordination policies to talk about space

- ReSpecT observation predicates are extended with physical & virtual space:

$$\begin{aligned} \langle \textit{ObservationPredicate} \rangle &::= \langle \textit{EventView} \rangle \texttt{\_} \langle \textit{EventInformation} \rangle \\ \langle \textit{EventView} \rangle &::= \texttt{current} \mid \texttt{event} \mid \texttt{start} \\ \langle \textit{EventInformation} \rangle &::= \ldots \mid \texttt{place} \mid \texttt{node} \mid \ldots \end{aligned}$$

- three ReSpecT guard predicates are introduced:

$$\langle \textit{GuardPredicate} \rangle ::= \ldots \mid \texttt{at(} \langle \textit{Place} \rangle \texttt{)} \mid \texttt{near(} \langle \textit{Place} \rangle \texttt{,} \langle \textit{Radius} \rangle \texttt{)} \mid \texttt{on(} \langle \textit{Node} \rangle \texttt{)} \mid \ldots$$

ReSpecT Tuple Centres as Space-aware Media IV

Coordination policies: talking about space *(req. 4 – contd.)*

In the overall, ReSpecT is extended with space

- by introducing some spatial predicates and guards to get information about both tuple-centre and event physical & virtual positioning

observation `current_place(?Place)`, `start_place(?Place)`, `event_place(?Place)`, `current_node(?Node)`, `start_node(?Node)`, `event_node(?Node)`

guard `at(@Place)`, `near(@Place,@Radius)`, `on(@Node)`

ReSpecT Tuple Centres as Space-aware Media V

Capturing spatial events *(req. 5)*

6. a coordination medium should be able to capture spatial events, and to react appropriately
   - the ReSpecT *admissible (tuple centre) event* model is extended to include spatial events

$$
\begin{array}{rcl}
\langle \mathit{TCEvent} \rangle & ::= & \langle \mathit{OpEvent} \rangle \mid \langle \mathit{TEvent} \rangle \mid \langle \mathit{SEvent} \rangle \mid \ldots \\
\langle \mathit{SEvent} \rangle & ::= & \langle \mathit{SStartCause} \rangle \, , \langle \mathit{SEventCause} \rangle \, , \langle \mathit{SResult} \rangle \, , \\
 & & \langle \mathit{Time} \rangle \, , \langle \mathit{Space{:}Place} \rangle \\
\langle \mathit{SStartCause} \rangle & ::= & \langle \mathit{SOp} \rangle \, , \texttt{space} , \langle \mathit{TCId} \rangle \\
\langle \mathit{SEventCause} \rangle & ::= & \langle \mathit{SStartCause} \rangle \\
\langle \mathit{SOp} \rangle & ::= & \texttt{from(} \langle \mathit{Space} \rangle \, , \langle \mathit{Place} \rangle \texttt{)} \mid \texttt{to(} \langle \mathit{Space} \rangle \, , \langle \mathit{Place} \rangle \texttt{)} \\
\langle \mathit{SResult} \rangle & ::= & \langle \mathit{SOp} \rangle \, , \ldots
\end{array}
$$

ReSpecT Tuple Centres as Space-aware Media VI

Capturing spatial events *(req. 5 – contd.)*

- correspondingly, the ReSpecT *event descriptor* is extended, too

$$\langle Event \rangle \quad ::= \quad \langle Predicate \rangle \texttt{(} \langle Tuple \rangle \texttt{)} \mid \texttt{time(} \langle Time \rangle \texttt{)} \mid$$
$$\texttt{from(} \langle Space \rangle \texttt{,} \langle Place \rangle \texttt{)} \mid \texttt{to(} \langle Space \rangle \texttt{,} \langle Place \rangle \texttt{)} \mid \ldots$$

thus making it possible to specify reactions to the occurrence of *spatial events*

```
reaction(from(?Space,?Place), Guard, Body).
reaction(to(?Space,?Place), Guard, Body).
```

Spatial Observation Predicates

`current_place(@S,?P)` succeeds if P unifies with the position of the node which the tuple centre belongs to

`event_place(@S,?P)` succeeds if P unifies with the position of the node where the triggering event was originated

`start_place(@S,?P)` succeeds if P unifies with the position of the node where the event chain that led to the triggering event was originated

The node position can be specified as either

- `S=ph` its absolute physical position
- `S=ip` its IP number
- `S=dns` its domain name
- `S=map` its geographical location—as typically defined by mapping services like Google Maps
- `S=org` its organisational position—that is, a location within an organisation-defined virtual topology

Spatial Guard Predicates

`at(@S,@P)` succeeds when the tuple centre is currently executing at the position P, specified according to S

`near(@S,@P,@R)` succeeds when the tuple centre is currently executing at the position included in the spatial region with centre P and radius R, specified according to S

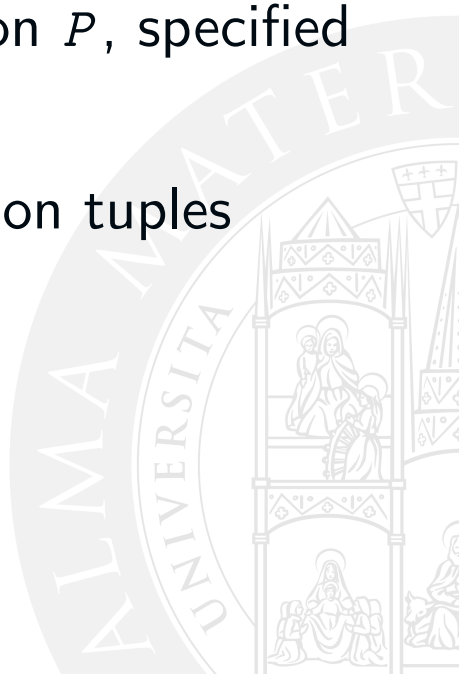
Spatial Event Descriptors

`from(?S,?P)` matches a spatial event raised when the device hosting the tuple centre starts moving from position P, specified according to S.

`to(?S,?P)` matches a spatial event raised when the device hosting the tuple centre stops moving and reaches position P, specified according to S.

As a result, the following are admissible reaction specification tuples dealing with spatial events:

```
reaction(from(?Space,?Place), Guards, Reactions).
reaction(to(?Space,?Place), Guards, Reactions).
```



Next in Line...

Focus on...

Situatedness & Coordination I

Situatedness...

- essentially, strict coupling with the environment
- technically, the ability to properly perceive and react to changes in the environment
- one of the most critical issues in distributed systems
  - conceptual clash between pro-activeness in process behaviour and reactivity w.r.t. environment change
- still one of the most critical issues for artificial intelligence & robotics

  [Suchman, 1987]

Situatedness & Coordination II

. . . & coordination

- essentially, situatedness concerns interaction between processes and the environment
- technically, situatedness can be conceived as a coordination problem
  - how to handle and govern interaction between pro-active processes and an ever-changing environment

Governing interaction

- intra-system interaction via coordination media as rulers of component-component interaction
- inter-system interaction via. . . ?
  - coordination media as rulers of component-environment interaction

Focus on...

Situating ReSpecT I

ReSpecT tuple centres for environment engineering

- distributed systems are immersed into an environment, and should be reactive to events of *any* sort
- also, coordination media should mediate any activity toward the environment, allowing for a fruitful interaction

$\Rightarrow$ ReSpecT tuple centres should be able to *capture general environment events*, and to generally *mediate process-environment interaction*

Situating ReSpecT II

Situating ReSpecT

- thus, *situating* the ReSpecT language basically means making ReSpecT capable of *capturing environment events*, and *expressing general MAS-environment interactions* [Casadei and Omicini, 2009]

$\Rightarrow$ ReSpecT captures, reacts to, and observes general environment events

$\Rightarrow$ ReSpecT can explicitly interact with the environment

ReSpecT Tuple Centres as Environment-aware Media I

Coordination policies: talking about environment

1. a coordination medium should allow coordination policies to talk about environment

- ReSpecT observation predicates are extended with environment

$$\begin{aligned} \langle \textit{ObservationPredicate} \rangle &::= \langle \textit{EventView} \rangle\_\langle \textit{EventInformation} \rangle \\ \langle \textit{EventView} \rangle &::= \texttt{current} \mid \texttt{event} \mid \texttt{start} \\ \langle \textit{EventInformation} \rangle &::= \ldots \mid \texttt{env} \end{aligned}$$

- two ReSpecT guard predicates are introduced:

$$\langle \textit{GuardPredicate} \rangle ::= \ldots \mid \texttt{from\_env} \mid \texttt{to\_env}$$

ReSpecT Tuple Centres as Environment-aware Media II

Coordination policies: talking about environment *(contd.)*

In the overall, ReSpecT is extended towards environment change

- by introducing some environment predicates and guards to get information about the environment status

observation
```
current_env(?Key,?Value),
start_env(?Key,?Value),
event_env(?Key,?Value),
```
guard
```
from_env, to_env
```

ReSpecT Tuple Centres as Environment-aware Media III

Capturing general environment events

2. a coordination medium should be able to capture environment events, and to react appropriately

- the ReSpecT *admissible (tuple centre) event* model is extended to include environment events

$$
\begin{array}{rcl}
\langle TCEvent \rangle & ::= & \langle OpEvent \rangle \mid \langle TEvent \rangle \mid \langle SEvent \rangle \mid \langle EEvent \rangle \ldots \\
\langle EEvent \rangle & ::= & \langle EStartCause \rangle \, , \langle EEventCause \rangle \, , \langle EResult \rangle \, , \\
 & & \langle Time \rangle \, , \langle Space{:}Place \rangle \\
\langle EStartCause \rangle & ::= & \langle EOp \rangle \, , \langle EResId \rangle \, , \langle TCId \rangle \\
\langle EDirectCause \rangle & ::= & \langle EStartCause \rangle \\
\langle EOp \rangle & ::= & \texttt{env}(\langle Key \rangle \, , \langle Value \rangle \,) \\
\langle EResult \rangle & ::= & \langle EOp \rangle \, , \ldots
\end{array}
$$

ReSpecT Tuple Centres as Environment-aware Media IV

Capturing general environment events *(contd.)*

- correspondingly, the ReSpecT *event descriptor* is extended, too

$$\begin{aligned}\langle \mathit{Event} \rangle \quad ::= \quad & \langle \mathit{Predicate} \rangle \texttt{(} \langle \mathit{Tuple} \rangle \texttt{)} \mid \texttt{time(} \langle \mathit{Time} \rangle \texttt{)} \mid \\ & \texttt{from(} \langle \mathit{Space} \rangle \texttt{,} \langle \mathit{Place} \rangle \texttt{)} \mid \texttt{to(} \langle \mathit{Space} \rangle \texttt{,} \langle \mathit{Place} \rangle \texttt{)} \mid \\ & \texttt{env(} \langle \mathit{Key} \rangle \texttt{,} \langle \mathit{Value} \rangle \texttt{)}\end{aligned}$$

making it possible to specify reactions to the occurrence of environment events

```
reaction(env(?Key,?Value), Guard, Body).
```

Causing Environment Change I

Environment manipulation

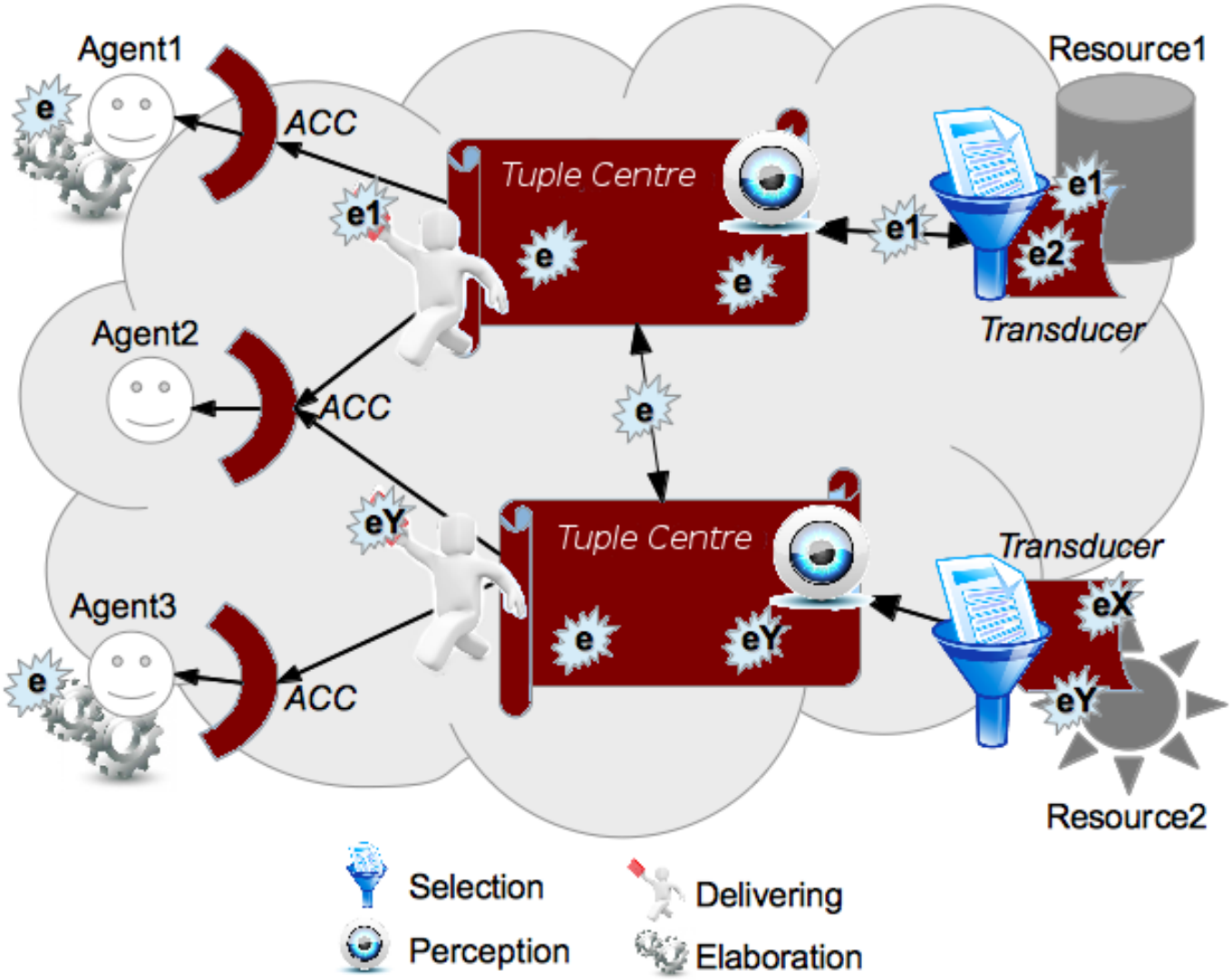
- source and target of a tuple centre event can be any external resource
- a suitable *identification* scheme – both at the syntax and at the infrastructure level – is introduced for environmental resources
- the ReSpecT language is extended to express explicit manipulation of environmental resources
- the body of a ReSpecT reaction can contain a *tuple centre predicate* of the form
  - $\langle EResId \rangle$ `?` `get`($\langle Key \rangle$,$\langle Value \rangle$)
    enabling a tuple centre to get properties of environmental resources
  - $\langle EResId \rangle$ `?` `set`($\langle Key \rangle$,$\langle Value \rangle$)
    enabling a tuple centre to set properties of environmental resources

Causing Environment Change II

Transducers

- specific environment events have to be translated into well-formed ReSpecT tuple centre events
- this should be done at the infrastructure level, through a general-purpose schema that could be specialised according to the nature of any specific resource
- a ReSpecT *transducer* is a component able to bring environment-generated events to a ReSpecT tuple centre (and back), suitably translated according to the general ReSpecT event model
- each transducer is specialised according to the specific portion of the environment it is in charge of handling—typically, the specific resource it is aimed at handling, like a temperature sensor, or a heater.

ReSpecT-TuCSoN Architecture

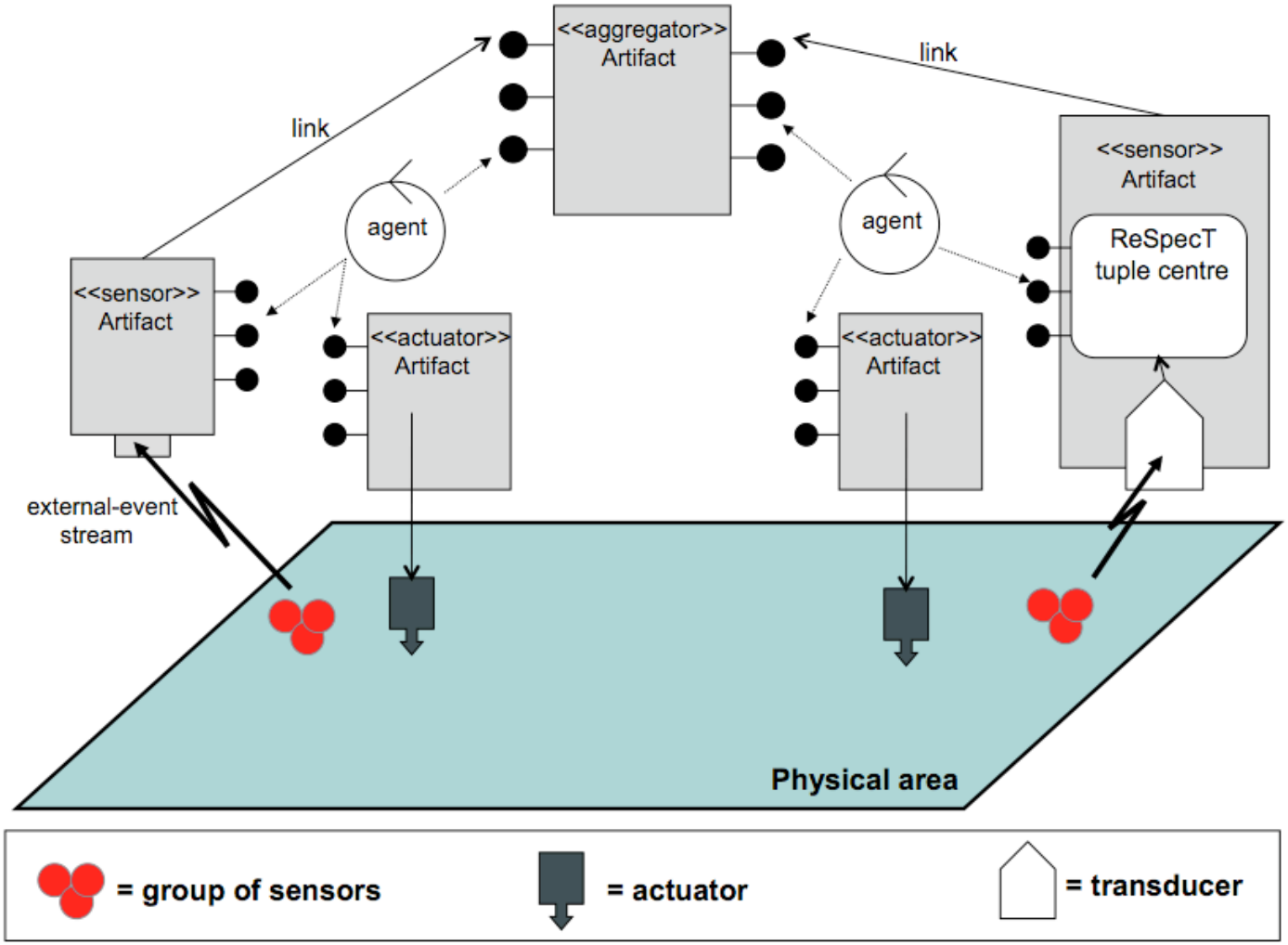


Focus on...

Controlling Environmental Properties of Physical Areas I

- a set of real *sensors* are used to measure some environmental property (for instance, temperature) within an area where they are located
- such information is then exploited to govern suitably placed *actuators* (say, heaters) that can affect the value of the observed property in the environment
- sensors are supposed to be cheap and non-smart, but provided with some kind of communication interface – either wireless or wired – that makes it possible to send streams of sampled values of the environmental property under observation
- accordingly, sensors are active devices, that is, devices *pro-actively* sending sensed values at a certain rate with no need of being asked for such data—this is what typically occurs in *pervasive computing* scenarios
- altogether, actuators and sensors are part of a distributed system aimed at controlling environmental properties (in the case study, temperature), which are affected by actuators based on the values measured by sensors and the designed control policies as well
- coordination policies can be suitably automated and encapsulated within coordination media working as environment artifacts controlling sensors and actuators

Case Study: ReSpecT-based Architecture



Case Study: Structure of Environment Artifacts

Environment artifacts are built based on of ReSpecT tuple centres:

- `<<sensor>> artifacts` wrapping real temperature sensors which perceive temperature of different areas of the room
- `<<actuator>> artifacts` wrapping actuators, which act as heating devices so as to control temperature
- `<<aggregator>> artifact` provides an aggregated view of the temperature values perceived by sensors spread in the room since it is linked to `<<sensor>> artifacts`:
  - `<<sensor>> artifacts` update tuples on `<<aggregator>> artifact` through *linkability*

Case Study: Sensor Artifacts

```
%(1)
reaction( env(temperature, Temp), from_env, (
     event_time(Time), event_source(sensor(Id)),
     out(sensed_temperature(Id,Temp,Time)),
     tc_aggr@node_aggr ? out(sensed_temperature(Id,Temp)) )
).
%(2)
reaction( out(sensed_temperature(_,Temp,_)), from_tc, (
     in(current_temperature(_)),
     out(current_temperature(Temp)) )
).
```

Behaviour

- reaction (1) is triggered by external events generated by a temperature sensor
- reaction (2) updates current temperature

Case Study: Aggregator Artifacts

```
%(4)
reaction( out(sensed_temperature(Id,Temp)), from_tc, (
      in(total_temperature(OldTotalTemp),
      in(sensed_temperature(Id,OldTemp)),
      TotalTemp is OldTotalTemp - OldTemp + Temp,
      out(total_temperature(TotalTemp),
      rd(number_of_sensors(SensorNo),
      AvgTemp is TotalTemp / SensorNo,
      in(average_temp(_)), out(average_temp(AvgTemp)) )
).
```

Behaviour

- reaction (4) keeps track of the current state of the average temperature

Case Study: Agents

Observable behaviour

Agents are goal-oriented and proactive processes that control temperature of the room

1. get local information from sensor
   `tc_sens@node_i ? rd(current_temperature(Temp_i))`
2. get global information from aggregator
   `tc_aggr@node_aggr ? rd(average_temp(AvgTemp))`
3. deliberate action by determining `TempVar` based on `Temp_i` and `AvgTemp`
4. act upon actuators (if `TempVar`$\neq 0$)
   `tc-heat_i@node_i ? out(change_temperature(TempVar))`

Case Study: Actuator Artifacts

```
%(3)
reaction( out(change_temperature(TempVar)), from_agent,
    actuator_i ? set(temp_inc,TempVar)
).
```

Behaviour

When the controller agent deliberate an increment in the temperature

- a `tc-heat_i@node_i ? out(change_temperature(TempVar))` reaches the actuator artifact
- by reaction (3), a suitable signal is sent to the actuator, through the suitably-installed transducer

Next in Line...

Focus on...

Dining Philos in ReSpecT: How to Fix Starvation?

! the problem is *time*: no one keeps track of time here, and starvation is a matter of time

? how can we handle time here? Is synchronisation not enough for the purpose?

✗ of course not: to avoid problems like starvation, we need the ability of defining *time-dependent* coordination policies

What is the solution?

$\Rightarrow$ in order to define time-dependent coordination policies, a time-aware coordination medium is needed

ReSpecT Events Revised

By recalling [Omicini, 2007], we may note that a ReSpecT event is not limited to be any ReSpecT (either coordination or meta-coordination) primitive:

$$\langle SimpleTCEvent \rangle ::= \langle SimpleTCPredicate \rangle \ (\langle Tuple \rangle) \mid \texttt{time}(\langle Time \rangle)$$

1. a `time(`T`)` event is generated by the ReSpecT VM when its *local time* (a relative notion of time measured starting from ReSpecT VM boot) reaches T
2. then, any reaction having that time value as a triggering event is triggered and (iff guards evaluate to `true`) executed

ReSpecT Guards Revised

Correspondingly, other guard predicates are provided to manage time-related aspects:

`before(`*Time*`)` $\iff$ the triggering event occurred before *Time*

`after(`*Time*`)` $\iff$ the triggering event occurred after *Time*

`between(`*Time*`,`*Time'*`)` $\iff$ `after(`*Time*`)` & `before(`*Time'*`)`

Timed Dining Philosophers: Philosopher

```
philosopher(I,J) :-
    think,                          % thinking
    table ? in(chops(I,J)),         % waiting to eat
    eat,                            % eating
    table ? out(chops(I,J)),        % waiting to think
!,  philosopher(I,J).
```

With respect to Dining Philosopher's protocol...

...this is left *unchanged* — and this is very convenient!

Timed Dining Philos: table ReSpecT Code

```
reaction( out(chops(C1,C2)), (operation, completion), (      % (1)
    in(chops(C1,C2)) )).
reaction( out(chops(C1,C2)), (operation, completion), (      % (1')
    in(used(C1,C2,_)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (       % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (       % (3)
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                  % (4)
    in(chop(C1)), in(chop(C2)), out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                          % (5)
    rd(required(C,C2)), in(chop(C)), in(chop(C2)), out(chops(C,C2)) )).
reaction( out(chop(C)), internal, (                          % (5')
    rd(required(C1,C)), in(chop(C1)), in(chop(C)), out(chops(C1,C)) )).
reaction( in(chops(C1,C2)), (operation, completion), (       % (6)
    current_time(T), rd(max eating time(Max)), T1 is T + Max,
    out(used(C1,C2,T)),
    out_s(time(T1),(in(used(C1,C2,T)), out(chop(C1)), out(chop(C2)))) )).
```

Timed Dining Philosophers in ReSpecT: Results

Results

protocol	no deadlock
protocol	fairness
protocol	trivial philosopher's interaction protocol
tuple centre	shared resources handled properly
tuple centre	no starvation =)

Let's try!

Checkout example `TDiningPhilosophersTest` in package `alice.tucson.examples.timedDiningPhilos`

Time-Aware Coordination Medium

- time-awareness is an essential feature to enable situatedness, that is the ability of a system to recognize the *temporal* environment in which it lives, thus to react properly to its contingencies and dynamism
- the other fundamental feature is Space-awarness, that is the ability to recognize the *spatial* environment, properly expressing, generating and perceiving *topology-related* aspects

More details in

Timed ReSpecT [Omicini et al., 2005]

Situated ReSpecT [Casadei and Omicini, 2009]

Focus on...

ReSpecT A&A Inspectability

Another extension to the original ReSpecT, provided by its A&A interpretation, is in ReSpecT events observability (*inspectability* of artifacts in the A&A terminology).

Observation predicates

In fact, the body of a ReSpecT reaction is not limited to exploit only ReSpecT primitives and Prolog predicates, but can use the so-called observation predicates

$\langle \textit{ObservationPredicate} \rangle ::= \langle \textit{EventView} \rangle\_\langle \textit{EventInformation} \rangle$

$\langle \textit{EventView} \rangle ::= \texttt{current} \mid \texttt{event} \mid \texttt{start}$

$\langle \textit{EventInformation} \rangle ::= \texttt{predicate} \mid \texttt{tuple} \mid \texttt{source} \mid \texttt{target} \mid \texttt{time}$

Observability Semantics

Any combination of the following is admissible in ReSpecT, following formal grammar of ⟨*ObservationPredicate*⟩ in previous slide

⟨*EventView*⟩ — to inspect the *events chain* triggering the executing reaction:

- `current` — access the ReSpecT event currently under processing
- `event` — access the ReSpecT event which is the direct cause of the event triggering the reaction
- `start` — access the ReSpecT event which is the prime cause of the event triggering the reaction

⟨*EventInformation*⟩ — to inspect all the data ReSpecT events make observable:

- `predicate` — the ReSpecT primitive causing the event
- `tuple` — the logic tuple argument of the `predicate`
- `source` — who performed the `predicate`
- `target` — who is directed to the `predicate`
- `time` — when the `predicate` was issued

Focus on...

Situating TuCSoN I

TuCSoN coordination for environment engineering

- distributed systems are *situated*—that is, immersed into an environment, and reactive to events of *any* sort
- thus, coordination media are required to mediate any activity toward the environment, allowing for a fruitful interaction

$\Rightarrow$ ReSpecT tuple centres are able to *capture general environment events*, and to generally *mediate process-environment interaction*

Situating TuCSoN II

Situating TuCSoN

- thus, *situating* TuCSoN basically means making it capable of *capturing environment events*, and *expressing general MAS-environment interactions*

  [Casadei and Omicini, 2009, Omicini and Mariani, 2013]

$\Rightarrow$ the TuCSoN middleware and the ReSpecT language

- capture, react to, and observe general environment events
- explicitly interact with the environment

Dealing with Environment Change I

Environment manipulation

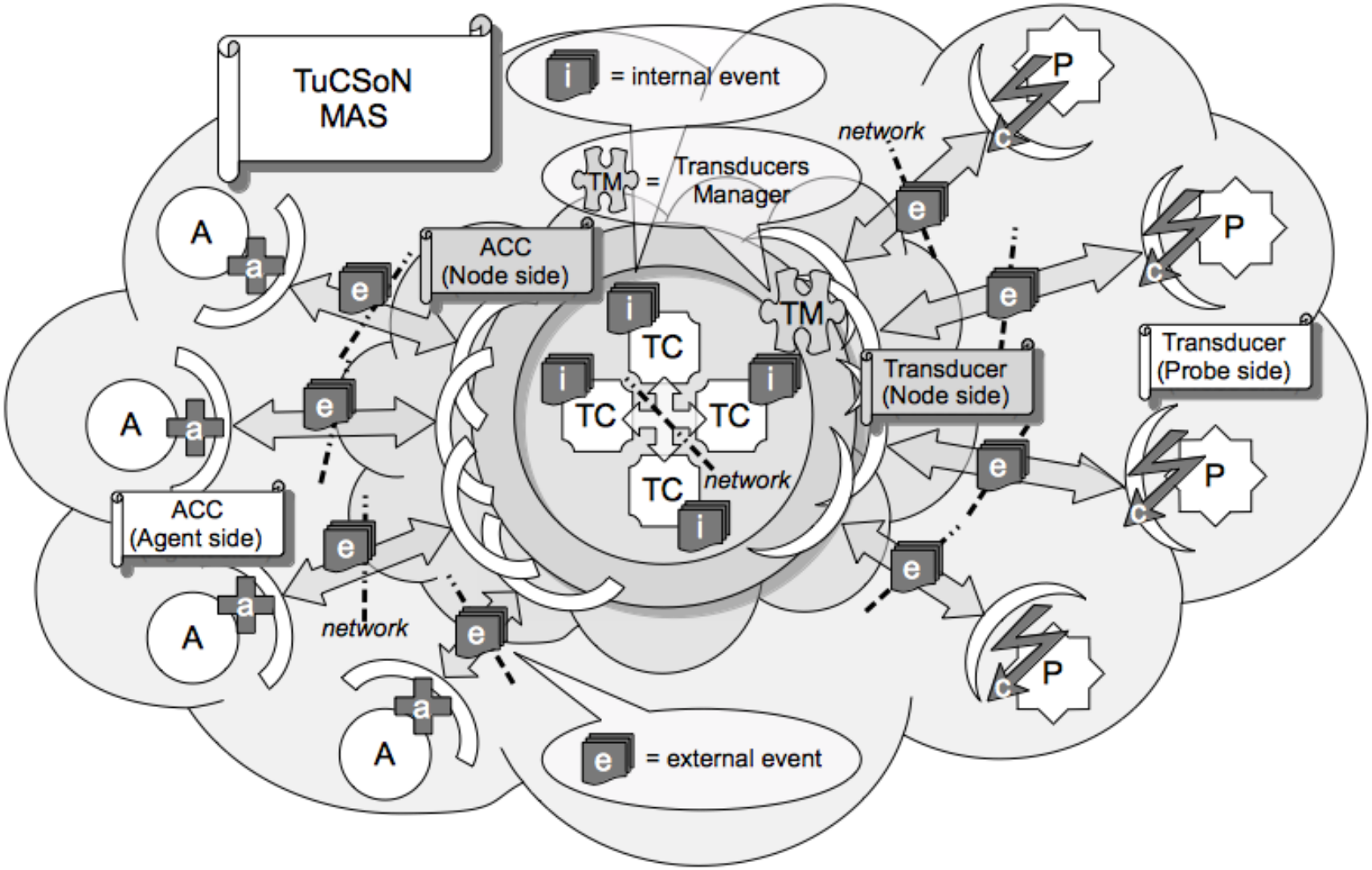
- source and target of a tuple centre event can be any external resource
- a suitable *identification* scheme – both at the syntax and at the infrastructure level – is introduced for environmental resources
- the coordination language is extended to express explicit manipulation of environmental resources
- new *tuple centre predicates* are introduced, whose form is
  - $\langle EResId \rangle$ `?` `get(`$\langle Key \rangle$`,`$\langle Value \rangle$`)`
    enabling a tuple centre to get properties of environmental resources
  - $\langle EResId \rangle$ `?` `set(`$\langle Key \rangle$`,`$\langle Value \rangle$`)`
    enabling a tuple centre to set properties of environmental resources

Dealing with Environment Change II

Transducers

- specific environment events have to be translated into well-formed ReSpecT tuple centre events
- this is to be done at the infrastructure level, through a general-purpose schema that could be specialised according to the nature of any specific resource
- a *transducer* is a component able to bring environment-generated events to a ReSpecT tuple centre (and back), suitably translated according to the general ReSpecT event model
- each transducer is specialised according to the specific portion of the environment it is in charge of handling—typically, the specific resource it is aimed at handling, like a temperature sensor, or a heater.

TuCSoN Situated Architecture



Example & Further References

Let's try

Check out example `Thermostat` in package `ds.lab.tucson.respect.situatedness`

```
java -cp libs/tucson.jar:libs/2p.jar:bin
ds.lab.tucson.respect.situatedness.Thermostat
```

More on transducers & situatedness

- papers
  - http://link.springer.com/chapter/10.1007/978-3-319-11692-1_9
  - http://ceur-ws.org/Vol-1260/paper11.pdf
- how-to
  http:
  //apice.unibo.it/xwiki/bin/download/TuCSoN/Documents/situatednesspdf.pdf

Focus on...

What About Coordination & Space?

- the availability of a plethora of mobile devices is pushing forward the needs for space-awareness of computations and systems
  - often essential to establish which tasks to perform, which goals to achieve, and how
- more generally, spatial issues are fundamental in many sorts of complex software systems, including adaptive, and self-organising ones [Beal, 2010]

Space-aware Coordination

In most of the application scenarios where *situatedness* plays an essential role, computation and coordination are required to be *space aware*

Situatedness & Awareness I

Situatedness

A space-aware coordination abstraction should at any time be associated to an *absolute positioning*, both *physical* and *virtual*

In fact

- *software* abstractions may move along a *virtual space* – typically, the network – which is usually *discrete*
- whereas *hardware* devices move through a *physical space*, which is mostly *continuous*

! however, software abstractions may also be hosted by mobile physical devices, thus *share* their motion

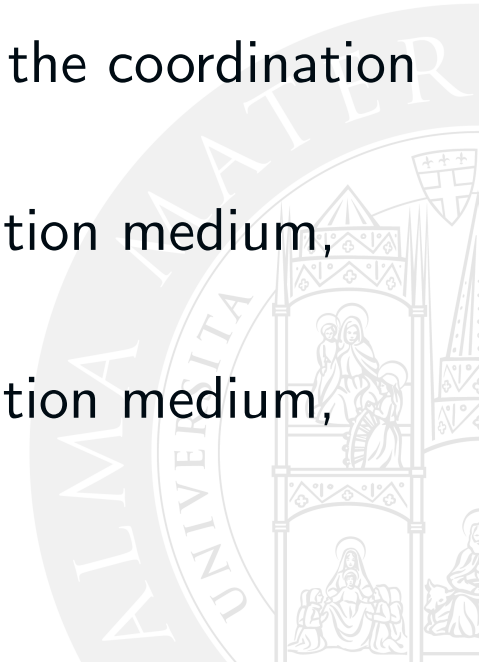
Situatedness & Awareness II

Awareness

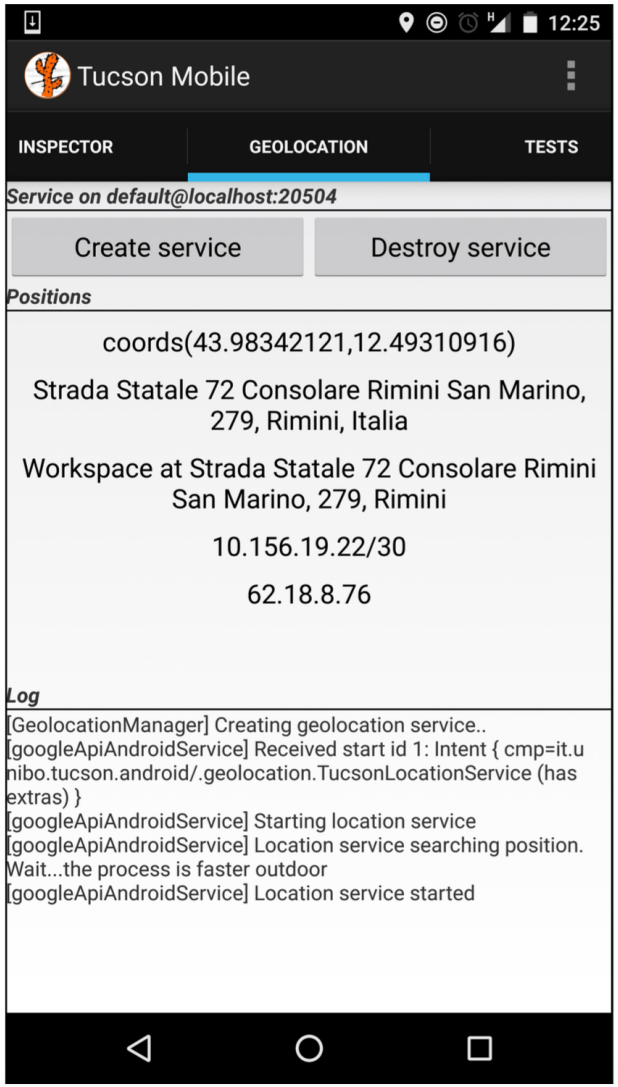
The position of the coordination medium should be available to the *coordination laws* it contains in order to make them capable of *reasoning about space* — thus, to implement space-aware coordination laws

Also, space has to be embedded into the working cycle of the coordination medium

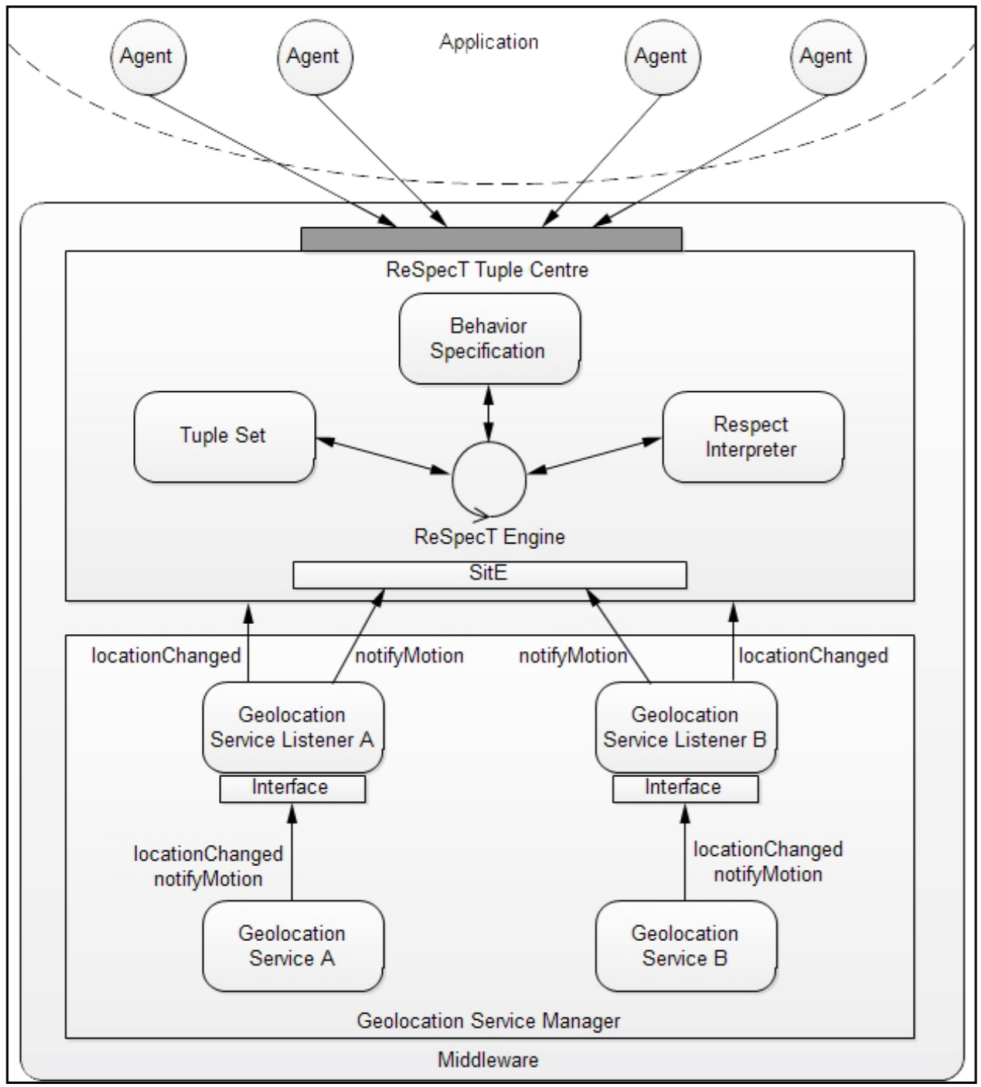
- ✓ a spatial event should be *generated* within a coordination medium, conceptually corresponding to *changes in space*
- ✓ then, such events should be *captured* by the coordination medium, and used to *activate* space-aware coordination laws



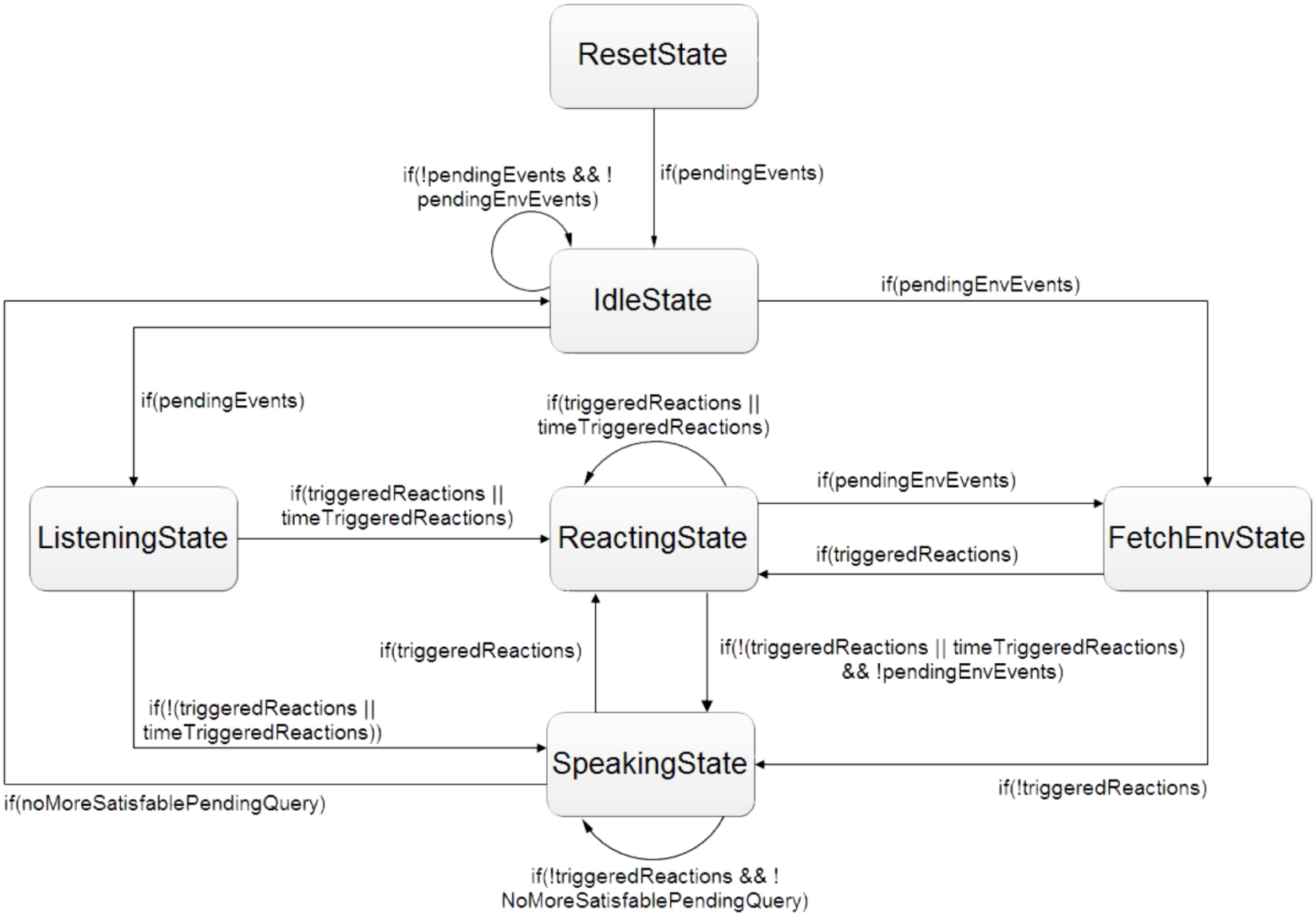
Space-aware Middleware: TuCSoN on Android I



Space-aware Middleware: TuCSoN on Android II



Space-aware Middleware: TuCSoN on Android III



Space-aware Middleware: TuCSoN on Android IV

Codebase

`https://bitbucket.org/smariani/tucsonandroid/overview`

References I

Arbab, F. (2004).
Reo: A channel-based coordination model for component composition.
Mathematical Structures in Computer Science, 14:329–366.

Baldauf, M., Dustdar, S., and Rosenberg, F. (2007).
A survey on context-aware systems.
International Journal of Ad Hoc and Ubiquitous Computing, 2(4):263–277.

Beal, J. (2010).
A basis set of operators for space-time computations.
In *Proceedings of the 2010 Fourth IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshop (SASOW 2010)*, pages 91–97, Washington, DC, USA. IEEE Computer Society.

Casadei, M. and Omicini, A. (2009).
Situated tuple centres in ReSpecT.
In Shin, S. Y., Ossowski, S., Menezes, R., and Viroli, M., editors, *24th Annual ACM Symposium on Applied Computing (SAC 2009)*, volume III, pages 1361–1368, Honolulu, Hawai'i, USA. ACM.

References II

Dastani, M., Arbab, F., and de Boer, F. S. (2005).
Coordination and composition in multi-agent systems.
In Dignum, F., Dignum, V., Koenig, S., Kraus, S., Singh, M. P., and Wooldridge, M. J., editors, *4rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005)*, pages 439–446, Utrecht, The Netherlands. ACM.

Denti, E., Natali, A., and Omicini, A. (1998).
On the expressive power of a language for programming coordination media.
In *1998 ACM Symposium on Applied Computing (SAC'98)*, pages 169–177, Atlanta, GA, USA. ACM.
Special Track on Coordination Models, Languages and Applications.

Omicini, A. (2002).
Towards a notion of agent coordination context.
In Marinescu, D. C. and Lee, C., editors, *Process Coordination and Ubiquitous Computing*, chapter 12, pages 187–200. CRC Press, Boca Raton, FL, USA.

References III

Omicini, A. (2007).
Formal ReSpecT in the A&A perspective.
Electronic Notes in Theoretical Computer Science, 175(2):97–117.
5th International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA'06), CONCUR'06, Bonn, Germany, 31 August 2006. Post-proceedings.

Omicini, A. and Denti, E. (2001).
From tuple spaces to tuple centres.
Science of Computer Programming, 41(3):277–294.

Omicini, A. and Mariani, S. (2013).
Coordination for situated MAS: Towards an event-driven architecture.
In Moldt, D. and Rölke, H., editors, *International Workshop on Petri Nets and Software Engineering (PNSE'13)*, volume 989 of *CEUR Workshop Proceedings*, pages 17–22. Sun SITE Central Europe, RWTH Aachen University.
Joint Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE'13) and the International Workshop on Modeling and Business Environments (ModBE'13), co-located with the 34th International Conference on Application and Theory of Petri Nets and Concurrency (Petri Nets 2013). Milano, Italy, 24–25 June 2013. Invited paper.

References IV

Omicini, A., Ricci, A., and Viroli, M. (2005).
Time-aware coordination in ReSpecT.
In Jacquet, J.-M. and Picco, G. P., editors, *Coordination Models and Languages*, volume 3454 of *LNCS*, pages 268–282. Springer-Verlag.
7th International Conference (COORDINATION 2005), Namur, Belgium, 20–23 April 2005. Proceedings.

Omicini, A., Ricci, A., and Viroli, M. (2007).
Timed environment for Web agents.
Web Intelligence and Agent Systems, 5(2):161–175.

Suchman, L. A. (1987).
Plans and Situated Actions: The Problem of Human-Machine Communication.
Cambridge University Press, New York, NYU, USA.

Weyns, D., Omicini, A., and Odell, J. (2007).
Environment as a first-class abstraction in multi-agent systems.
Autonomous Agents and Multi-Agent Systems, 14(1):5–30.
Special Issue on Environments for Multi-agent Systems.

References V

Zambonelli, F., Castelli, G., Ferrari, L., Mamei, M., Rosi, A., Di Marzo Serugendo, G., Risoldi, M., Tchao, A.-E., Dobson, S., Stevenson, G., Ye, Y., Nardini, E., Omicini, A., Montagna, S., Viroli, M., Ferscha, A., Maschek, S., and Wally, B. (2011).
Self-aware pervasive service ecosystems.
Procedia Computer Science, 7:197–199.
Proceedings of the 2nd European Future Technologies Conference and Exhibition 2011 (FET 11).

Programming Interaction with ReSpecT

Distributed Systems / Technologies

Sistemi Distribuiti / Tecnologie

Stefano Mariani Andrea Omicini

{s.mariani, andrea.omicini}@unibo.it

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

Alma Mater Studiorum – Università di Bologna a Cesena

Academic Year 2017/2018