Coordination in Situated Systems

Engineering MAS Environment in TuCSoN

Stefano Mariani, Andrea Omicini

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

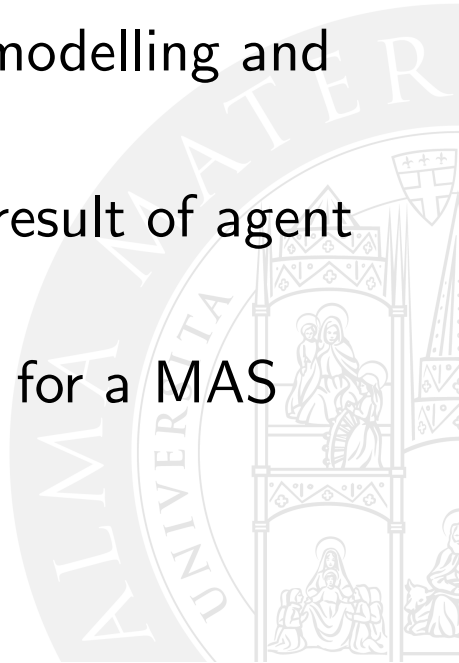
Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Alma Mater Studiorum—Università di Bologna, Italy

Talk @ IDCS 2014
International Conference on
Internet and Distributed Computing Systems
Amantea, CS, Calabria, Italy
22nd-24th September 2014

Outline

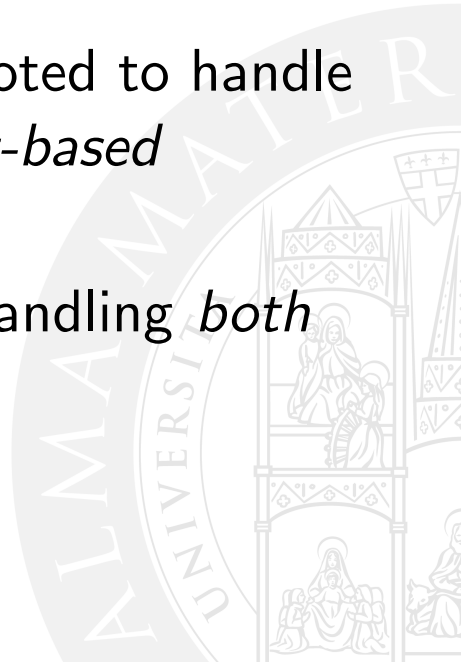
Context

- Multi-agent systems (MAS) are an expressive paradigm for the engineering of distributed systems [Jennings, 2001, Jennings, 2000]
- Agents are not the only fundamental bricks for multi-agent systems (MAS) [Omicini et al., 2008]
- Environment too is an essential abstraction for MAS modelling and engineering [Weyns et al., 2007]
- This means that in MAS things happen not just as a result of agent *actions*, but also because of environment change
- These are the two sources of events and dependencies for a MAS



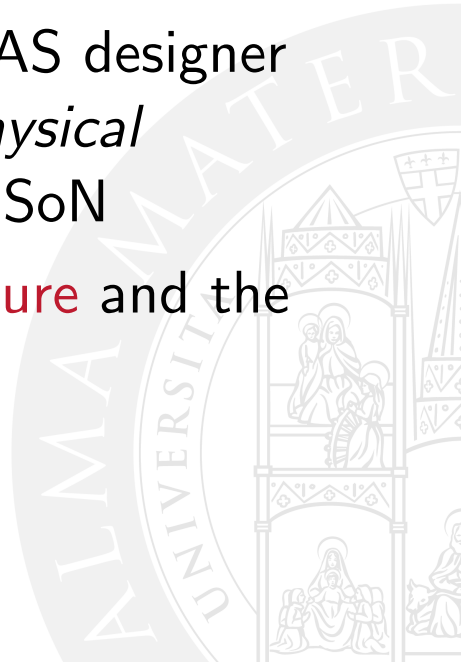
Motivation

- Both agent-agent (social) and agent-environment (situated) *dependencies* are sources of complexity in MAS
- Since coordination aims at managing dependencies [Malone and Crowston, 1994], it could be used to deal with both social and situated dependencies
- Till now, coordination artefacts have been mostly devoted to handle social interaction—however, as a form of *environment-based coordination* [Omicini et al., 2004]
- It is then quite natural to see coordination artefacts handling *both* social and situated interaction in a uniform way [Omicini and Mariani, 2013]



Goals

- We focus on the TuCSoN coordination middleware [Omicini and Zambonelli, 1999], and show how it supports environment engineering in MAS by providing *coordination artefacts* [Omicini et al., 2004] to handle situated interaction
- We focus on situatedness by describing the steps a MAS designer should follow to *effectively engineer computational/physical environments* supporting *agent situated action* in TuCSoN
- The contribution is meant to be on both the architecture and the method sides

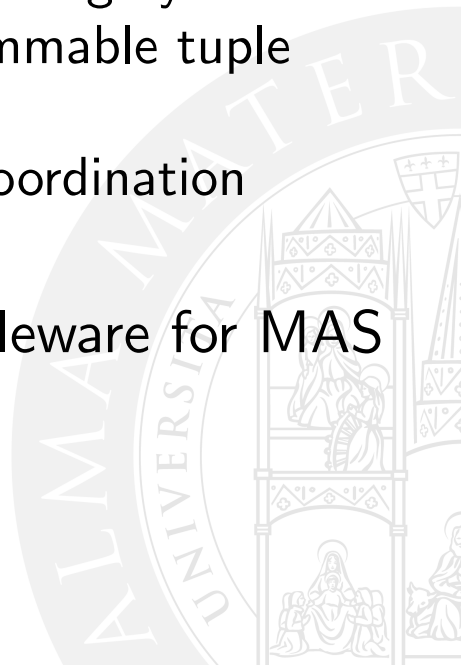


Outline

What is TuCSoN?

- TuCSoN [Omicini and Zambonelli, 1999] is a *tuple-based coordination model* [Gelernter and Carriero, 1992] for open, distributed MAS, providing ReSpecT *tuple centres* [Omicini and Denti, 2001] as its coordination artefacts, which means that
  - a TuCSoN-coordinated MAS is made by agents interacting by exchanging *tuples* via *tuple centres*, which are programmable tuple spaces [Denti et al., 1997]
  - TuCSoN tuple centres contain the policies for MAS coordination written in the ReSpecT language
- The TuCSoN model is provided as a Java-based middleware for MAS engineering at

`http://tucson.unibo.it`



TuCSoN Architecture for Situatedness I

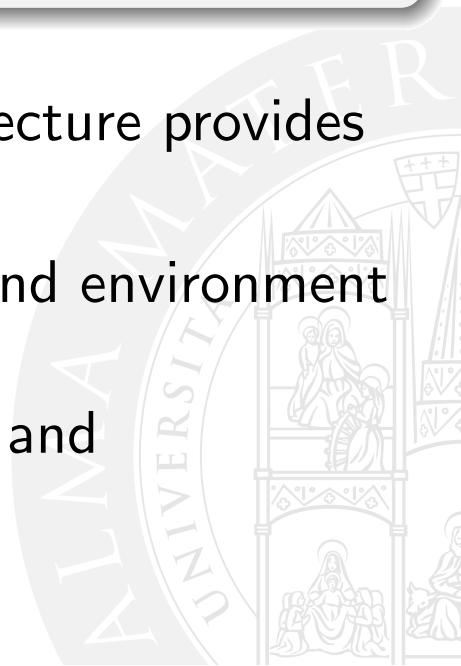
Artefacts

- According to the A&A meta-model [Omicini et al., 2008], a MAS is made of *agents* and *artefacts*
- where artefacts are the *reactive* entities embodying the *functions* of the MAS

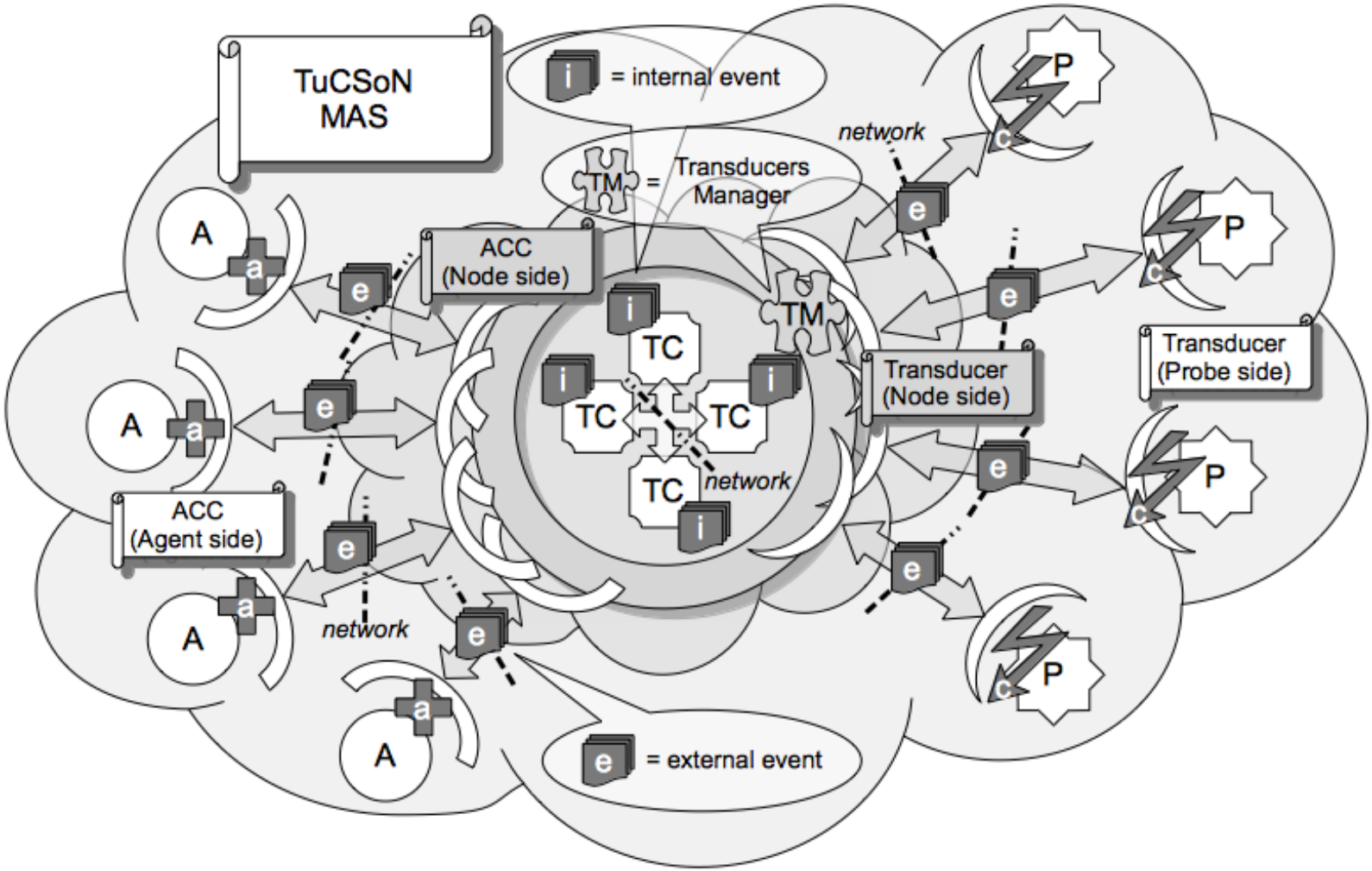
- Inspired by the A&A meta-model, the TuCSoN architecture provides two sorts of artefacts

  boundary artefacts — to handle both agent *activity* and environment *change* within the MAS

  coordination artefacts — to govern both *agent-agent* and *agent-environment* interaction



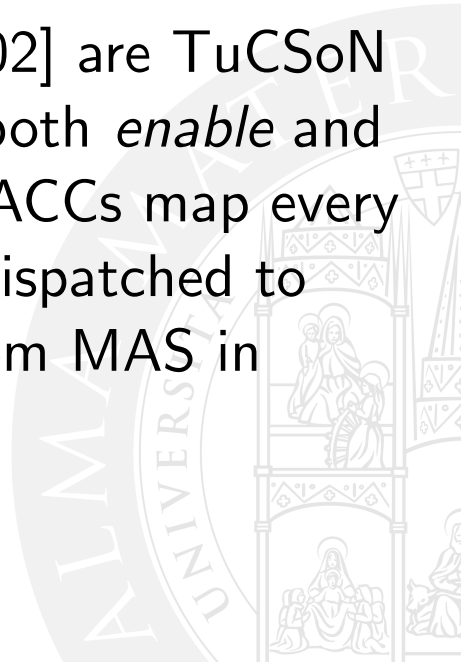
TuCSoN Architecture for Situatedness II



TuCSoN Architecture for Situatedness III

agents — Any computational entity willing to exploit TuCSoN coordination services. Agents should request and obtain an ACC from the TuCSoN node. *Any* action from *any* agent towards the MAS – either social or situated – is then *mediated* by its associated ACC.

ACC — *Agent coordination contexts* [Omicini, 2002] are TuCSoN boundary artefacts devoted to agents. ACC both *enable* and *constraint* agents interactions: in particular, ACCs map every agent operation into events *asynchronously* dispatched to tuple centres. ACCs thus *decouple* agents from MAS in control, reference, space, and time.



TuCSoN Architecture for Situatedness IV

probes — Environmental resources in TuCSoN are called *probes*. They are dealt with as sources of perceptions (aka *sensors*) or makers of actions (aka *actuators*) in a *uniform* way. Probes do not directly interact with the MAS, but through *mediation* of their associated transducer.

transducers — TuCSoN *transducers* [Casadei and Omicini, 2009] are the boundary artefacts devoted to probes. Each probe is assigned to a transducer, which is *specialised* to handle events from that probe, and to act on probes through *situation* operations. Transducers thus *decouple* in control, reference, space and time probes from tuple centres.

TuCSoN Architecture for Situatedness V

events — TuCSoN adopts and generalises the ReSpecT event model. Events are the run-time data structure representing both activities and change in a uniform way. ACC and transducers translate *external events* (activities and change) into *internal events* (*operations* and *situations*) that tuple centres can handle to implement the policies required for MAS coordination

tuple centres — TuCSoN architectural component implementing coordination artefacts, thus in charge of managing *dependencies*. As such, they are meant to *govern* both social and situated interactions [Omicini and Mariani, 2013]. By adopting ReSpecT tuple centres, TuCSoN relies on *(i)* the ReSpecT language to program coordination laws, and *(ii)* the ReSpecT situated event model to implement events

TuCSoN Event Model I

$\langle Event \rangle$	::=	$\langle StartCause \rangle$, $\langle Cause \rangle$, $\langle Evaluation \rangle$
$\langle StartCause \rangle$, $\langle Cause \rangle$	::=	$\langle Activity \rangle$ | $\langle Change \rangle$, $\langle Source \rangle$, $\langle Target \rangle$, $\langle Time \rangle$, $\langle Space{:}Place \rangle$
$\langle Source \rangle$, $\langle Target \rangle$	::=	$\langle AgentId \rangle$ | $\langle CoordArtefactId \rangle$ | $\langle EnvResId \rangle$ | $\perp$
$\langle Evaluation \rangle$	::=	$\perp$ | $\{\langle Result \rangle\}$

Table: ReSpecT situated *event model*.

$\langle Activity \rangle$	::=	$\langle Operation \rangle$ | $\langle Situation \rangle$
$\langle Operation \rangle$	::=	`out(` $\langle Tuple \rangle$ `)` | (`in` | `rd` | `no` | `inp` | `rdp` | `nop`) `(` $\langle Template \rangle$ [`,` $\langle Term \rangle$] `)`
$\langle Situation \rangle$	::=	`getEnv(` $\langle Key \rangle$ `,` $\langle Value \rangle$ `)` | `setEnv(` $\langle Key \rangle$ `,` $\langle Value \rangle$ `)`
$\langle Change \rangle$	::=	`env(` $\langle Key \rangle$ `,` $\langle Value \rangle$ `)` | `time(` $\langle Time \rangle$ `)` |
		`from(` $\langle Space \rangle$ `,` $\langle Place \rangle$ `)` | `to(` $\langle Space \rangle$ `,` $\langle Place \rangle$ `)`

Table: ReSpecT *triggering events*.

TuCSoN Event Model II

Summing up

In a TuCSoN-coordinated MAS

- ACCs and transducers represent agents and environment, respectively, in the MAS, by translating activities and changes in a common event model
- Tuple centres deals with both social and situated dependencies by making it possible to program the coordination of all sort of event in a uniform way

In essence, this means that we need both a technology and a methodology to coherently exploit *event-driven programming* within an *agent-oriented programming* framework

TuCSoN Event Model III

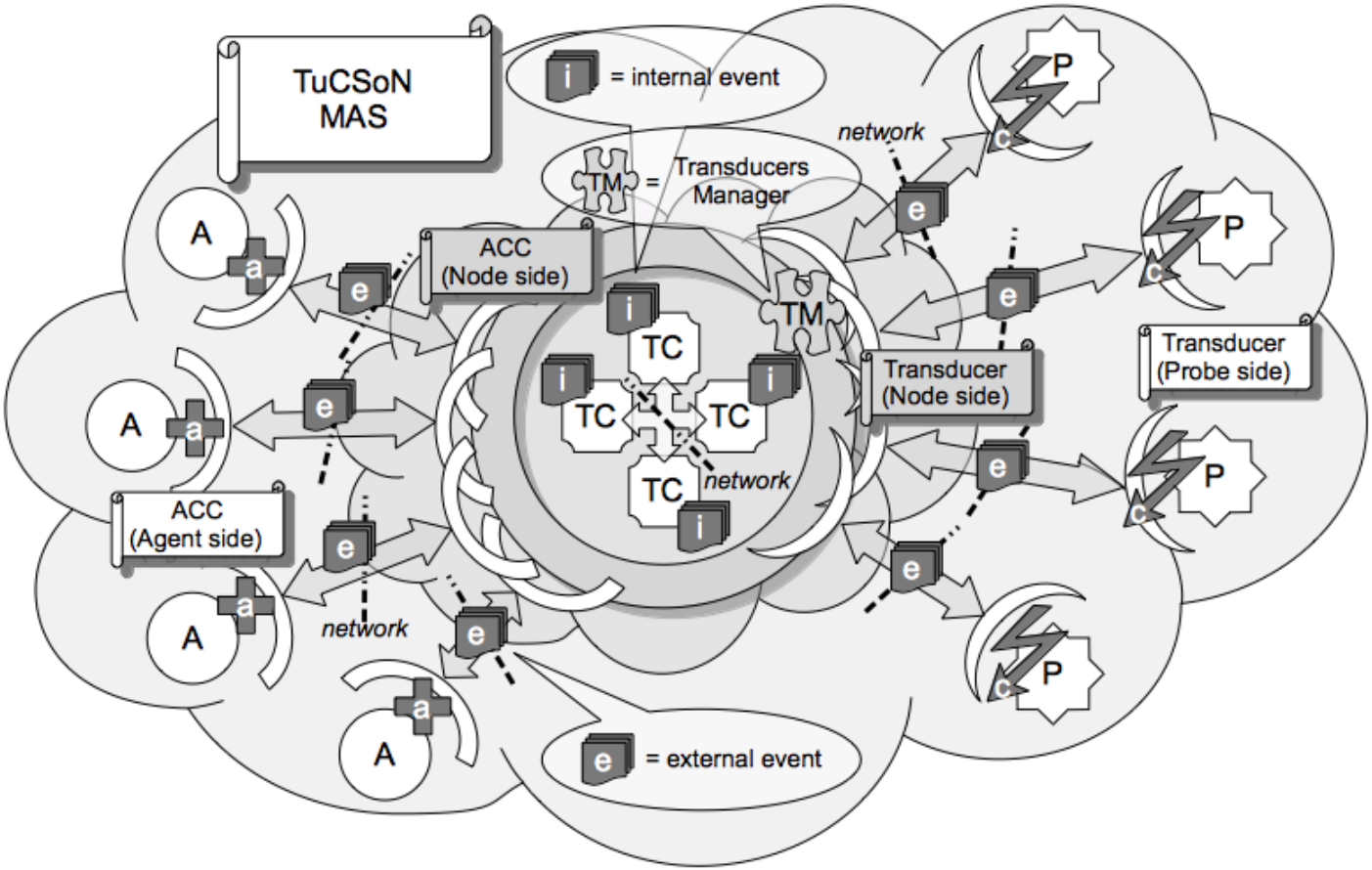
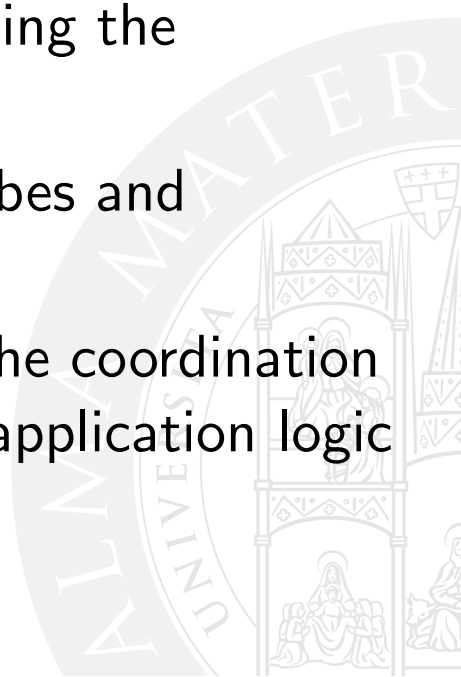


Figure: In TuCSoN, both social (agent-agent) and situated (agent-environment) interactions are *mediated* by ReSpecT tuple centres.

Outline

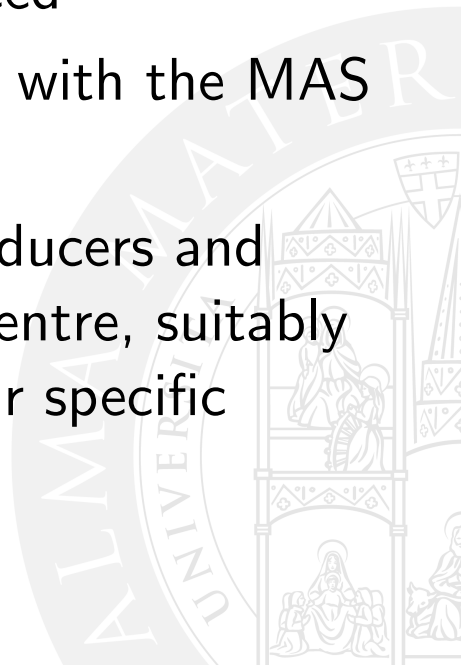
General Method for Environment Engineering

1. Implement probes—sensors and actuators. Typically, this does not require implementing, e.g., software drivers for the probe: designers can simply wrap existing drivers in a Java class implementing the `ISimpleProbe` interface, then interact with TuCSoN transducers
2. Implement transducers associated to probes by extending the TuCSoN `AbstractTransducer` class
3. Configure the transducer manager, responsible for probes and transducers association and lifecycle management
4. Program tuple centres using ReSpecT implementing the coordination policies that, along with TuCSoN agents, embed the application logic



Scenario

- A situated intelligent thermostat is in charge of keeping a room temperature between 18 and 22 degrees
- To this end, it interacts with a sensor and an actuator: the former is queried by the thermostat to perceive the temperature, whereas the latter is prompted to change the temperature upon need
- Both the sensor and the actuator, as probes, interface with the MAS through one transducer each
- To promote distribution of the application logic, transducers and thermostat are associated each with their own tuple centre, suitably programmed through ReSpecT reactions handling their specific interactions with the MAS

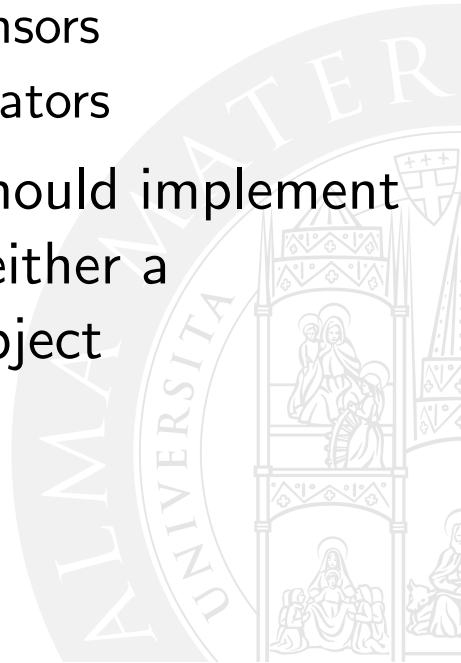


1) Implement Probes

- Implement the `ISimpleProbe` interface:

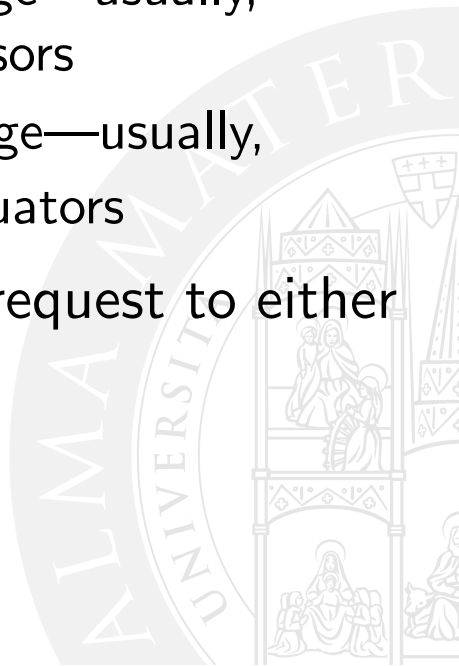
  `getIdentifier` — retrieving probe ID
  `getTransducer` — retrieving associated transducer
  `setTransducer` — associating an existing transducer to the probe
  `readValue` — to *perceive* the probe—mandatory for sensors
  `writeValue` — to *act* on the probe—mandatory for actuators

- In particular, methods `readValue` and `writeValue` should implement the logic required to interact with the actual probe—either a computational environmental resource or a physical object



2) Implement Transducers I

- An abstract Java class is provided for extension: `AbstractTransducer`, implementing `TransducerStandardInterface`
  - getEnv — to *sense* an environmental property change—usually, implemented by transducers assigned to sensors
  - setEnv — to *effect* an environmental property change—usually, implemented by transducers assigned to actuators
- Such methods should actually dispatch to probes the request to either sense/effect an environmental property change



2) Implement Transducers II

Middleware support

Both methods are *automatically* called by TuCSoN middleware whenever

- an event generated by an environmental property change is raised by the associated probe—`notifyEnvEvent` method in `TransducerStandardInterface`, to be called by MAS programmers
- an event generated by an environmental property change is raised by the associated tuple centre—`notifyOutput` method in `TransducerStandardInterface`[a]

[a] Automatically called by TuCSoN middleware in response to ReSpecT *situation activities*—see ⟨*Situation*⟩ in Table 2.

3) Configure the Transducer Manager I

- Once both probes and transducers are implemented, MAS designers should exploit TuCSoN `TransducersManager` services to register such components and to associate them:

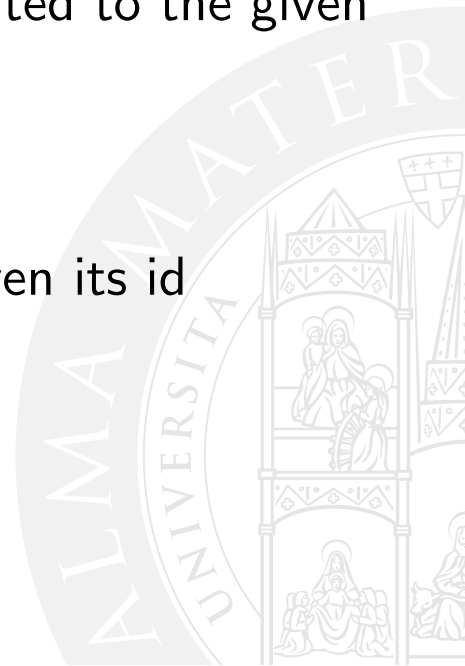
  `createTransducer` — to create a new transducer associated to the given probe and bound to the given tuple centre

  `addProbe` — to attach a probe to a given transducer

  `removeProbe` — to detach a probe from its transducer

  `getTransducer` — to retrieve a transducer's reference given its id

  `stopTransducer` — to destroy a given transducer



3) Configure the Transducer Manager II

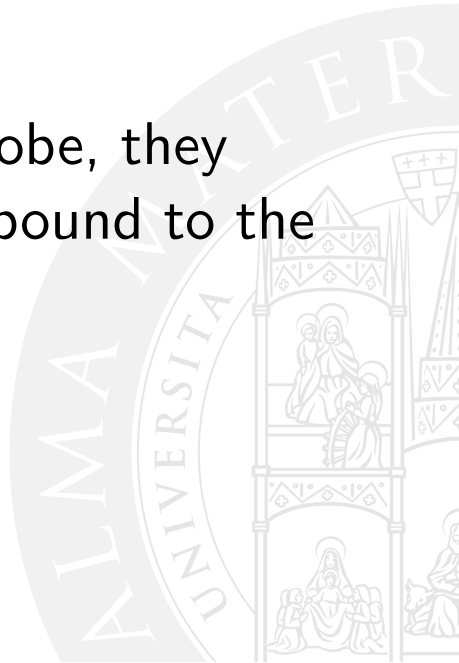
Run-time adaptiveness

- Notice to enable *dynamic* and *distributed addition/removal* of transducers and probes, as well as *run-time change* of their associations, all the transducer manager services are also available via TuCSoN coordination operations.
- In particular, TuCSoN agents may (remotely) emit special tuples in the built-in '`$ENV`' tuple centre, available in any TuCSoN node
- Syntax of such tuples can be found in TuCSoN official guide

`http://www.slideshare.net/andreaomicini/the-tucson-coordination-model-technology-a-guide`

4) Program Tuple Centres I

- Agents and probes (environment) are connected via ReSpecT tuple centres, programmed via the ReSpecT language
- In fact, agents and probes – or better, ACCs and transducers – do not directly interact: all the interactions happen through coordination operations provided by the TuCSoN middleware
- Therefore, whenever agents need to interact with a probe, they perform a coordination operation on the tuple centre bound to the transducer responsible for that probe



4) Program Tuple Centres II

```
1  reaction(
2    in(sense(temp(T))), // agent request
3    (operation, invocation),
4    sensor@localhost:20504 ? getEnv(temp, T) // perception request
5  ).
6  reaction(
7    getEnv(temp, T), // perception reply
8    (from_env, completion), // environment filter
9    out(sense(temp(T)))
10 ).
```

Figure: Stripped-down version of the code from `sensorSpec.rsp` in package `alice.tucson.examples.situatedness` within current TuCSoN distribution (`TuCSoN-1.10.7.0208`). 'sensor' is the probe ID of the probe target of the situation operation request: the id of its transducers is automatically retrieved by TuCSoN middleware at run-time, hence transducer mediation is transparent to the ReSpecT programmer.

4) Program Tuple Centres III

- ReSpecT pattern essentially valid for any situated interaction:
  - reaction $1-5$ maps agents coordination operations requests (external events) into situation operations commands (internal events)
  - reaction $6-10$ maps situation operation replies (from probes, external events) into coordination operations outcomes (internal events)

Supporting situatedness

MAS designers explicitly exploit ReSpecT situated event model to support situatedness of interactions, *binding together events* coming from the agent through its ACC with events going toward the environment through its transducer—and, dually, from the environment toward the agents.

The Whole Picture I

```
1  /* Start perception-reason-action loop */
2  LogicTuple template;
3  ITucsonOperation op;
4  int temp;
5  LogicTuple action = null;
6  for (int i = 0; i < Thermostat.ITERS; i++) {
7    /* Perception */
8    template = LogicTuple.parse("sense(temp(_))");
9    op = acc.in(sensorTc, template, null); // see line 2 in Figure 2
10   if (op.isResultSuccess()) {
11     temp = op.getLogicTupleResult().getArg(0).getArg(0).intValue();
12     /* Reason */
13     if ((temp >= Thermostat.LOW) && (temp <= Thermostat.HIGH)) {
14       continue;
15     } else if (temp < Thermostat.LOW) {
16       action = LogicTuple.parse("act(temp(" + ++temp + "))");
17     } else if (temp > Thermostat.HIGH) {
18       action = LogicTuple.parse("act(temp(" + --temp + "))");
19     }
20     /* Action */
21     // 'act' ReSpecT reactions are similar to those in Figure 2
22     acc.out(actuatorTc, action, null);
23   }
24 }
```

Figure: Stripped-down version of the code from `Thermostat.java`. Notice the thermostat interacts solely with TuCSoN tuple centres, being transducers (thus probes) interactions transparently delegated to the TuCSoN middleware—through the ReSpecT reactions in Figure 2.

The Whole Picture II

- The last code snippet in Figure 3 is meant to show how the application logic – the thermostat – is linked to the "situatedness machinery"
- In particular, line 9 shows TuCSoN coordination operation invocation causing ReSpecT reactions in Figure 2 to trigger, stimulating the sensor probe through its transducer—*transparently* to the designer of the application logic
- Same sort of transparency is provided to ReSpecT programmers, as they need not to know the internal machinery of probes but just transducer API, as well as to probes programmers, since they only deal with `ISimpleProbe` and `TransducerStandardInterface` API.

The Whole Picture III

Separation of concerns

This promotes and supports a clear separation of concerns between specialised programmers, such that

- *application logic* (agent) programmers
- *coordination* (ReSpecT) programmers
- *environment* (probes and transducers) programmers

each may focus on their task, just relying on the fact that all programmers will use the same TuCSoN API.

Outline

Conclusion

- *Agent-oriented frameworks* can be effective in the engineering of complex distributed systems by providing a coherent framework for event-driven programming—in particular when dealing with environment engineering in *situated systems*
- Agent coordination models, typically exploited for governing *social interaction*, could extend their reach to deal with *situated interaction*, by providing suitable abstractions to handle all *sources of change* in a MAS – agents and environment – through a uniform event model
- In this paper
  - we describe the architectural approach to situatedness adopted by the TuCSoN *coordination middleware*
  - we discuss the basic steps of a methodology allowing *agent-oriented engineering* and *event-driven programming* to fruitfully coexist in the engineering of situated MAS

References I

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.

Denti, E., Natali, A., and Omicini, A. (1997).
Programmable coordination media.
In Garlan, D. and Le Métayer, D., editors, *Coordination Languages and Models*, volume 1282 of *LNCS*, pages 274–288. Springer-Verlag.

Gelernter, D. and Carriero, N. (1992).
Coordination languages and their significance.
Communications of the ACM, 35(2):97–107.

Jennings, N. R. (2000).
On agent-based software engineering.
Artificial Intelligence, 117(2):277–296.

Jennings, N. R. (2001).
An agent-based approach for building complex software systems.
Communications of the ACM, 44(4):35–41.

References II

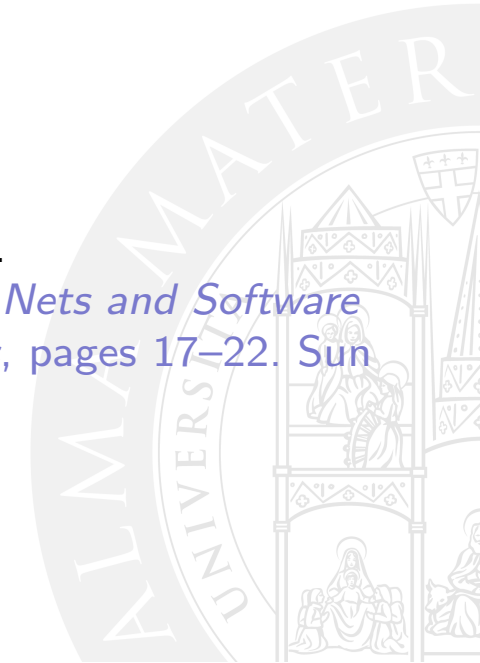
Malone, T. W. and Crowston, K. (1994).
The interdisciplinary study of coordination.
ACM Computing Surveys, 26(1):87–119.

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.

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.

Omicini, A., Ricci, A., and Viroli, M. (2008).
Artifacts in the A&A meta-model for multi-agent systems.
Autonomous Agents and Multi-Agent Systems, 17(3):432–456.



References III

Omicini, A., Ricci, A., Viroli, M., Castelfranchi, C., and Tummolini, L. (2004).
Coordination artifacts: Environment-based coordination for intelligent agents.
In Jennings, N. R., Sierra, C., Sonenberg, L., and Tambe, M., editors, *3rd international Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004)*, volume 1, pages 286–293, New York, USA. ACM.

Omicini, A. and Zambonelli, F. (1999).
Coordination for Internet application development.
Autonomous Agents and Multi-Agent Systems, 2(3):251–269.

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

URLs

Slides

- On APICe

  http://apice.unibo.it/xwiki/bin/view/Talks/SituatedcoordIdcs2014

- On SlideShare

  http://www.slideshare.net/andreaomicini/coordination-in-situated-systems-engineering-mas-environment-in-tucson

Article

- On APICe

  http://apice.unibo.it/xwiki/bin/view/Publications/SituatedcoordIdcs2014

- On SpringerLink

  http://link.springer.com/10.1007/978-3-319-10422-5_9

Coordination in Situated Systems

Engineering MAS Environment in TuCSoN

Stefano Mariani, Andrea Omicini

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

Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Alma Mater Studiorum—Università di Bologna, Italy

Talk @ IDCS 2014
International Conference on
Internet and Distributed Computing Systems
Amantea, CS, Calabria, Italy
22nd-24th September 2014