Advanced Coordination Techniques

Experiments with TuCSoN and ReSpecT

Stefano Mariani   Andrea Omicini
{s.mariani, andrea.omicini}@unibo.it

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

Faculté d'informatique – Université de Namur
Thursday, April 28th, 2016

Outline

Outline

Issues in Concurrent / Distributed Systems

- Concurrency / Parallelism
  - multiple independent activities / loci of control
  - active simultaneously
  - processes, threads, actors, active objects, agents, ...
- Distribution
  - activities running on different and heterogeneous execution contexts (virtual machines, devices, ...)
- Social interaction
  - *dependencies* among activities
  - collective goals involving activities coordination / cooperation
- Situated interaction
  - interaction with *environmental resources* (computational or physical)
  - interaction within the *time-space fabric*

(Non) Algorithmic Computation

What is a component in a distributed system?

- A computational abstraction characterised by
  - an independent *computational* activity
  - *I/O* capabilities

$\Rightarrow$ Two orthogonal dimensions
  - *computation*
  - interaction

Beyond Turing Machines

- Turing's *choice machines* and *unorganised machines* [WG03]
- Wegner's *Interaction Machines* [GSW06]

Compositionality vs. Non-compositionality

Compositionality

Sequential composition $P1; P2$

$$behaviour(P1; P2) = behaviour(P1) + behaviour(P2)$$

Non-compositionality

Interactive composition $P1 \mid P2$

$$behaviour(P1 \mid P2) = behaviour(P1) + behaviour(P2) + interaction(P1, P2)$$

$\Rightarrow$ Interactive composition is *more* than the sum of its parts

Non-compositionality: Issues

- Compositionality vs. formalisability
  - a formal model is required for stating any compositional property
  - however, formalisability does not require compositionality, and does not imply *predictability*
  - partial formalisability may allow for proof of properties, and for partial predictability
- Emergent behaviours
  - fully-predictable / formalisable systems *do not allow* by definition for emergent behaviours
- Formalisability vs. *expressiveness*
  - Less / more formalisable systems are (*respectively*) more / less expressive in terms of potential behaviours

Basic Engineering Principles

- Abstraction
  - problems should be represented / faced at the right level of abstraction
  - resulting abstractions should be expressive enough to capture the most relevant problems
  - conceptual integrity
- Locality & encapsulation
  - design abstractions should embody the solutions corresponding to the domain entities they represent
- Run-time vs. design-time abstractions
  - incremental change / evolution
  - on-line engineering [FG04]
  - (cognitive) self-organising systems [Omi12a]

Outline

What is a Coordination Model?

Coordination model as a glue

"A coordination model is the glue that binds separate activities into an ensemble" [GC92]

Coordination model as an agent interaction framework

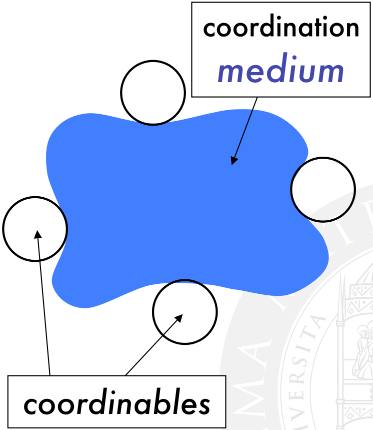
"A coordination model provides a framework in which the interaction of active and independent entities called agents can be expressed" [Cia96]

Coordination: Sketching a Meta-model [Cia96]

The *medium of coordination*

"Fills" the *interaction space*

- enables / promotes / governs the admissible / desirable / required interactions among the interacting entities (*coordinables*)
- according to coordination laws
    - enacted by the behaviour of the medium
    - defining the semantics of coordination



Requirements for a Coordination Model

What do we ask to a coordination model?

- To provide high-level abstractions and suitably expressive mechanisms for distributed system engineering
- To intrinsically *add properties* to systems *independently* of components
  - e.g. robustness, flexibility, control, intelligence, adaptiveness, self-organisation, ...

Classes of Coordination Models

Control-oriented vs. Data-oriented Models [PA98]

Control-oriented Focus on the *acts* of communication

Data-oriented Focus on the *information* exchanged during communication

- Several surveys, no time enough here
- Are these really *classes*?
  - actually, better to take this as a criterion to *observe* coordination models, rather than to *separate* them

Control-oriented Models

Which abstractions?

- Producer-consumer pattern
- Point-to-point communication
- *Coordinator* as the ruler of the space of interactions
- Coordination as configuration of topology

Which systems?

- Fine-grained granularity
- Fine-tuned control
- Good for small-scale, *closed* systems

Data-oriented Models

- Information-based design & interaction (thus, coordination)
- Possible *spatio-temporal uncoupling*
- Different sort of control over interacting components (*governing vs. commanding*)
- Examples
  - Gamma / chemical coordination
  - LINDA & friends / tuple-based coordination

An Evolutionary Pattern?

Paradigms of sequential programming

1. Imperative programming with “goto”
2. Structured programming (procedure-oriented)
3. Object-oriented programming (data-oriented)

Paradigms of coordination programming

1. Message-passing coordination
2. Control-oriented coordination
3. Data-oriented coordination

Outline

Outline

LINDA [Gel85]: Main Features

tuples ordered collection of information chunks, possibly heterogeneous in sort

generative communication until explicitly withdrawn, tuples live independently w.r.t. their producers, and are equally accessible to all the coordinables, but are bound to none

associative access tuples are accessed based on their content & structure, rather than by name, address, or location

suspensive semantics coordination operations (e.g., `out`, `in`, `rd`) may be suspended based on unavailability of matching tuples, and be woken up when such tuples become available

LINDA: Associative Access

- *Synchronisation* based on tuple content & structure
  - absence / presence of tuples with a *given (partial)* content / structure determines the behaviour of the coordinables (then, of the system)
  - based on *tuple templates* & *matching mechanism*

⇒ *Information-driven coordination*
  - patterns of coordination based on data / information availability

- *Reification*
  - making events become *tuples*
  - grouping classes of events with tuple syntax, and accessing them via tuple templates

LINDA: Suspensive Semantics

- `in` & `rd` primitives have a *suspensive semantics*
  - the coordination medium makes the primitive *wait* in case a matching tuple is not found, and *wakes* it up when such a tuple is found
  - the coordinable invoking the suspensive primitive is *expected to* wait for its successful completion

⇒ *Twofold wait*

in the coordination medium
: the operation is first (possibly) suspended, then (possibly) served

in the coordinated entity
: the invocation *may* cause a wait-state in the invoker ⇒ hypothesis on the *internal behaviour* of the coordinable

Dining Philos in LINDA

Philosopher using chopstick pairs: chops(I,J)

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

Issues

+ fairness, no deadlock
+ trivial philosopher's interaction protocol
– shared resources not handled properly (chops should be independent)
– starvation still possible (eating forever)

Dining Philos in LINDA: Where is the Problem?

- ✗ The behaviour of the coordination medium is *fixed* once and for all
  - ⇒ coordination problems that fits it are solved satisfactorily, those that do not fit *are not*
- ✗ Introducing novel primitives, e.g., bulk primitives (e.g., `in_all`, `rd_all`), is not a *general-purpose* solution
  - ⇒ adding ad hoc primitives does not solve the problem once and for all
- ✗ As a result, the coordination load is typically charged upon coordination entities
  - ⇒ this *does not* follow basic software engineering principles, like encapsulation, locality, separation of concerns

Dining Philos in LINDA: Solution?

✓ Making the behaviour of the coordination medium adaptable to the coordination problem at hand
  ⇒ in principle, *all* coordination problems may fit *some* admissible behaviour of the coordination medium
  ⇒ no need to either add new *ad hoc* primitives, or change the semantics of the old ones

⇒ This way, coordination media could *encapsulate* solutions to coordination problems
  - represented in terms of *coordination policies*
  - enacted in terms of *coordinative behaviour* (of the coordination media)

! What is needed is a way to *define* the behaviour of a coordination medium
  ⇒ a general computational model for coordination media
  ⇒ along with a suitably expressive programming language to define their behaviour

Hybrid Coordination Models

- In a sense, we need to add a control-driven layer to a data-oriented coordination model

? Why not purely control driven, then?

  ! control-driven coordination does not fit information-driven contexts, e.g., Web-based ones — quite obviously

  ! they also have difficulties in dealing with autonomy [DAdB05], a fundamental feature of, e.g., multi-agent systems

⇒ We need *hybrid* coordination models

Towards Tuple Centres

- What should be added to a tuple-based model to make it hybrid, and how?
- 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
- Which new features?
  - ✓ ability to *define new coordinative behaviours* embodying required coordination policies
  - ✓ ability to *associate* coordinative behaviours to coordination events

Outline

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 coordination events [OD01a]

- 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, called reactions
- A *reaction specification language*, thus
  - ✓ enables definition of reactions...
  - ✓ ...and makes it possible to associate them to the events occurring in a tuple centre

Reactions

- Each reaction can
  - ✓ access and modify the current tuple centre state — e.g., adding or removing tuples
  - ✓ access the information related to the triggering event – e.g., the performing process, the primitive invoked, the tuple involved, etc. – which is made completely *observable*
  - ✓ invoke link primitives — coordination primitives whose target is another (*remote*) tuple centre

⇒ As a result, the semantics of the traditional coordination primitives – e.g., `out`, `rd`, `in` – is no longer constrained to be as simple as in the LINDA model

- instead, it can be made *as complex as required* by the specific application needs

Tuple Centre Working Cycle I

The main cycle of a tuple centre works as follows

1. when a primitive *invocation* reaches a tuple centre, all the corresponding reactions (if any) are *triggered*, and then executed atomically and transactionally in a *non-deterministic* order
2. once *all* the reactions have been executed, the primitive is served in the same way as in standard LINDA
3. upon *completion* of the invocation, the corresponding reactions (if any) are triggered, and then executed according to the aforementioned semantic
4. once all the reactions have been executed, the main cycle of a tuple centre may go on possibly serving another invocation

Tuple Centre Working Cycle II

As a result, tuple centres exhibit a couple of fundamental features

Tuple spaces as "empty" tuple centres

An *empty* behaviour specification *defaults* the behaviour of a tuple centre to that of a tuple space

Tuple centres still are tuple spaces

From the process' perspective, the result of the invocation of a tuple centre primitive is the sum of

- the effects of the primitive itself
- the effects of all the reactions it triggers

perceived altogether as a *single-step transition* of the tuple centre state

Tuple Centre's State vs. Process' Perception

- 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
  - thanks to *atomic* and *transactional* semantic of reactions execution
- Unlike a standard tuple space, the perceived transition of a tuple centre state can be made *as complex as needed*
  - ⇒ this enables a novel form of *decoupling*: the process' view of a tuple centre may be different from the actual state of the same tuple centre

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
  - *full observability* of events
  - the ability to *selectively react* to events
  - the ability to *program coordination rules* by manipulating the interaction space

Outline

Outline

Tuple Centres Spread over the Network (TuCSoN)

TuCSoN is a model for the coordination of *distributed processes*, as well as of autonomous agents [OZ99]

References

main page `http://tucson.unibo.it/`

Bitbucket `http://bitbucket.org/smariani/tucson/`

FaceBook `http://www.facebook.com/TuCSoNCoordinationTechnology`

Core Abstractions I

- TuCSoN agents are the *coordinables*
- ReSpecT tuple centres are the programmable *coordination media* [OD01b]
- TuCSoN nodes represent the basic *topological abstraction*, which host the tuple centres
  - ⇒ agents, tuple centres, and nodes have *unique identities* within a TuCSoN system
- Agents act on tuple centres by means of *coordination operations*, built out of the TuCSoN coordination language, as defined by the collection of TuCSoN coordination primitives

Core Abstractions II

- Agents may live *anywhere* on the network, and may interact with tuple centres hosted by *any reachable* TuCSoN node
- Agents can *move independently* of the device where they execute [OZ98], while tuple centres' *mobility depends* on their hosting device moving abilities

System view

Roughly speaking, a TuCSoN system is a collection of (mobile) agents and tuple centres, hosted on possibly mobile devices, *coordinating* in a (distributed) set of nodes

Nodes

- Each node within a TuCSoN system is *univocally identified* by the pair $< \mathit{NetworkId}, \mathit{PortNo} >$, where
  - *NetworkId* is the *IP number* of the device hosting the node
  - *PortNo* is the *TCP port number* where the TuCSoN *coordination service* listens to incoming requests

$\Rightarrow$ Correspondingly, the abstract syntax[1] of TuCSoN nodes identifiers is

netid : portno

(`localhost : 20504`)

[1] Actually, this is also the concrete syntax used by TuCSoN to parse nodes' IDs

Tuple Centres

- An *admissible name* for a tuple centre is *any Prolog-like*, first-order logic *ground term*[2] [Llo84]
- Each tuple centre is *uniquely* identified by its admissible name *associated* to the node identifier

⇒ Hence the TuCSoN *full name* of a tuple centre *tname* on a node *netid* : *portno* is

tname @ *netid* : *portno*

(`default @ localhost : 20504`)

[2] *Ground* roughly means "no variables"

Agents

- An *admissible name* for an agent is *any Prolog-like*, first-order logic ground term too
- When it *enters* a TuCSoN system, an agent is assigned a *universally unique identifier* (UUID)[3]

⇒ If an agent *aname* is assigned UUID *uuid*, its *full name* is

aname : uuid

(`stefano : 4baad505-ad2f-4ac4-b30b-bc3705a2c87a`)

[3] `http://docs.oracle.com/javase/7/docs/api/java/util/UUID.html`

Coordination Language

- TuCSoN *coordination operations* are built out of *coordination primitives* and of the communication languages:
  - the tuple language
  - the tuple template language
- In TuCSoN, both the tuple and the tuple template languages are *logic-based*, too
  - *any* first-order logic Prolog atom is an *admissible* TuCSoN tuple...
  - ...and an *admissible* TuCSoN tuple template
  - ⇒ the two languages *coincide*, thus

Coordination Operations

- Any TuCSoN *coordination operation* is invoked by a source agent on a target tuple centre, which is in charge of its execution

  invocation phase — the request of the agent reaches the tuple centre, decorated with information about the invocation

  completion phase — the response of the tuple centre goes back to the agent, including information about operation execution outcome

- The abstract syntax[4] of a coordination operation *op* invoked on a target tuple centre *tcid* is

$$\mathit{tcid}\ ?\ \mathit{op}$$

$$\mathit{tname}\ @\ \mathit{netid}\ :\ \mathit{portno}\ ?\ \mathit{op}$$

$$(\texttt{default @ localhost : 20504 ? out(t(hi))})$$

[4] Actually, this is also the concrete syntax used by TuCSoN to parse coordination operations, even inside ReSpecT reactions

Coordination Primitives

The TuCSoN *coordination language* provides the following 9 coordination primitives to build coordination operations:

- `out` to put a tuple in the target tuple centre
- `rd`, `rdp` to read a tuple matching a given tuple template in the target tuple centre
- `in`, `inp` to withdraw a tuple matching a given tuple template from the target tuple centre
- `no`, `nop` to check absence of tuples matching a given tuple template in the target tuple centre
- `get` to read all the tuples in the target tuple centre
- `set` to overwrite the set of tuples in the target tuple centre

Outline

Defaults I

- Many TuCSoN nodes can run on the *same* networked device, as long as each one is listening on a *different* TCP port
- The default TCP port number of TuCSoN is `20504`
  - ⇒ so, agents can invoke operations of the form

    tname @ netid ? op

    `(default @ localhost ? out(t(hi)))`
- *Any* other port can be used for a TuCSoN node (we will see how to change it in a few slides)
- The default tuple centre of a TuCSoN node is named `default`
  - ⇒ so, agents can invoke operations of the form

    @ netid : portno ? op

    `(@ localhost : 20504 ? out(t(hi)))`

Defaults II

- By combining defaults, the following invocations are also admissible for any TuCSoN agent running on a device *netid*:
  - ✓ *: portno ? op*
    invoking operation *op* on the `default` tuple centre of node *netid* : *portno*
  - ✓ *tname ? op*
    invoking operation *op* on the *tname* tuple centre of default node *netid* : 20504
  - ✓ *op*
    invoking operation *op* on the `default` tuple centre of default node *netid* : 20504

Global vs. Local Coordination Space

- TuCSoN global coordination space is defined by the collection of *all* the tuple centres available *on the network*, identified by their *full name*
  - ⇒ a TuCSoN agent running on *any* networked device has the whole TuCSoN global coordination space available for its coordination operations through invocations of the form

    `tname @ netid : portno ? op`
- TuCSoN local coordination space is defined by the collection of *all* the tuple centres available on *all* the TuCSoN nodes hosted by the *local* device — let `netid` be its network address
  - ⇒ a TuCSoN agent running on *the same* device (`netid`) can access the local coordination space by invoking operations of the form

    `tname : portno ? op`

Agent Coordination Context I

An Agent Coordination Context (ACC) [Omi02] is a *runtime* and *stateful* interface

- *enabling* an agent to execute coordination operations on the tuple centres of a specific *organisation*– -e.g., TuCSoN system
- *constraining* its admissible interactions

modelling RBAC in TuCSoN [ORV05a] — more on this in slide 192

Role of ACC

Along with tuple centres, ACC are the run-time abstraction that allows TuCSoN to *uniformly* handle coordination, organisation, and security issues

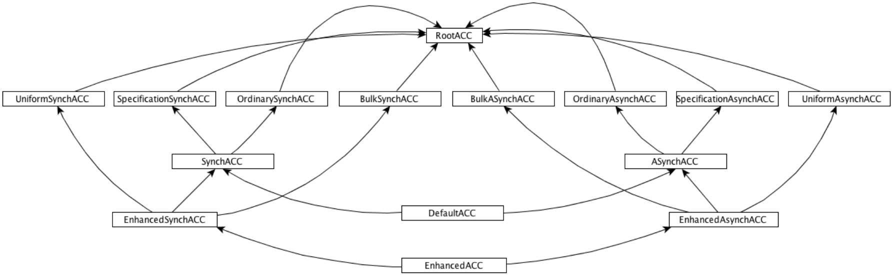
Agent Coordination Context II

`OrdinarySynchACC` enables interaction with the ordinary tuple space supporting a synchronous invocation semantics: whichever the coordination operation invoked (*either* suspensive or predicative), the agent *blocks* waiting for its completion

`SpecificationAsynchACC` enables interaction with the (ReSpecT) specification tuple space supporting an asynchronous invocation semantics: whichever the coordination operation invoked (*either* suspensive or predicative), the agent is *asynchronously notified* upon completion

... ...

Overview of TuCSoN ACCs



TuCSoN Middleware Overview

- TuCSoN is a Java-based middleware (Java 7 is enough)
- TuCSoN is also Prolog-based[5]: it is based on the tuProlog [DOR01] Java-based technology for
  - first-order logic tuples
  - primitives & identifiers
  - ReSpecT specification language & virtual machine
- TuCSoN middleware provides:
  - Java API for using TuCSoN coordination services from Java programs
    - package `alice.tucson.api.*` (mostly)
  - Prolog API for using TuCSoN coordination services from tuProlog programs

[5] Last digits in TuCSoN version number (`TuCSoN-1.12.0.0301`) are for the tuProlog version, hence tuProlog version 3.0.1 atm

TuCSoN Service

- A TuCSoN node can be started from a command prompt with:

```
java -cp tucson.jar:2p.jar alice.tucson.service.TucsonNodeService
    [-port portno]
```

- The node is in charge of
  - listening to incoming invocations of coordination operations
  - dispatching them to the target tuple centre
  - returning the operations completion to the source agent

Let's try!

1. Open a console, position yourself into the folder where `tucson` and `2p` jars are, then type the command above — on Windows, replace ":" with ";"
2. Try to launch another TuCSoN node on a different *portno*

TuCSoN CLI I

The Command Line Interpreter is a shell interface for humans

```
java -cp tucson.jar:2p.jar
    alice.tucson.service.tools.CommandLineInterpreter
    [-netid netid] [-port portno] [-aid CLIname]
```

Let's try!

In the console, type the command above giving *the same* `[-port portno]` given for TuCSoN installation — on Windows, replace ":" with ";"

TuCSoN CLI II

```
McGriddle:libs ste$ java -cp tucson.jar:2p.jar alice.tucson.service.tools.CommandLineInterpreter -netid localhost -portno 20504 -aid myCLI
[CommandLineInterpreter]: ------------------------------------------------------------------------------------
[CommandLineInterpreter]: Booting TuCSoN Command Line Intepreter...
[CommandLineInterpreter]: Version TuCSoN-1.12.0.0301
[CommandLineInterpreter]: ------------------------------------------------------------------------------------
[CommandLineInterpreter]: Fri Apr 15 15:40:16 CEST 2016
[CommandLineInterpreter]: Demanding for TuCSoN default ACC on port < 20504 >...
[CommandLineInterpreter]: Spawning CLI TuCSoN agent...
[CommandLineInterpreter]: ------------------------------------------------------------------------------------
[CLI]: CLI agent listening to user...
[CLI]: ?> help
[CLI]: ------------------------------------------------------------------------------------
[CLI]: TuCSoN CLI Syntax:
[CLI]:
[CLI]:         tcName@ipAddress:port ? CMD
[CLI]:
[CLI]: where CMD can be:
[CLI]:
[CLI]:         out(Tuple)
[CLI]:         in(TupleTemplate)
[CLI]:         rd(TupleTemplate)
[CLI]:         no(TupleTemplate)
[CLI]:         inp(TupleTemplate)
[CLI]:         rdp(TupleTemplate)
[CLI]:         nop(TupleTemplate)
[CLI]:         get()
[CLI]:         set([Tuple1, ..., TupleN])
[CLI]:         spawn(exec('Path.To.Java.Class.class')) | spawn(solve('Path/To/Prolog/Theory.pl', Goal))
[CLI]:         in_all(TupleTemplate, TupleList)
[CLI]:         rd_all(TupleTemplate, TupleList)
[CLI]:         no_all(TupleTemplate, TupleList)
[CLI]:         uin(TupleTemplate)
[CLI]:         urd(TupleTemplate)
[CLI]:         uno(TupleTemplate)
[CLI]:         uinp(TupleTemplate)
[CLI]:         urdp(TupleTemplate)
[CLI]:         unop(TupleTemplate)
[CLI]:         out_s(Event,Guard,Reaction)
[CLI]:         in_s(EventTemplate, GuardTemplate, ReactionTemplate)
[CLI]:         rd_s(EventTemplate, GuardTemplate, ReactionTemplate)
[CLI]:         inp_s(EventTemplate, GuardTemplate ,ReactionTemplate)
[CLI]:         rdp_s(EventTemplate, GuardTemplate, ReactionTemplate)
[CLI]:         no_s(EventTemplate, GuardTemplate, ReactionTemplate)
[CLI]:         nop_s(EventTemplate, GuardTemplate, ReactionTemplate)
[CLI]:         get_s()
[CLI]:         set_s([(Event1,Guard1,Reaction1), ..., (EventN,GuardN,ReactionN)])
[CLI]: ------------------------------------------------------------------------------------
[CLI]: ?>
```

TuCSoN CLI III

Let's try!

1. Try out coordination operations
2. Try to suspend the CLI. . .
3. . . . and to resume it =)
4. Try to access each other TuCSoN nodes tuple centres — the global *coordination space*
5. Try to spot *invocation* and *completion* phase of operations — look at TuCSoN node console log

TuCSoN Inspector

GUI tool to monitor the TuCSoN coordination space & ReSpecT VM

- to launch the Inspector

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

- available options are (also available from the GUI)

  `-aid` — the name to assign to the inspector

  `-netid` — the IP address where the TuCSoN node to inspect is reachable…

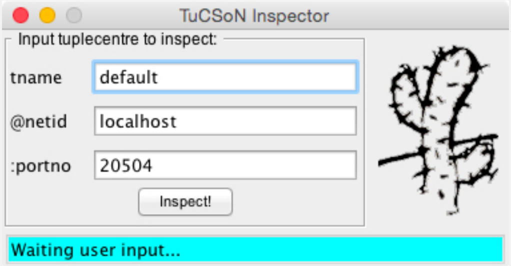
  `-portno` — …and its TCP listening port…

  `-tcname` — …and the name of the tuple centre to monitor

Using TuCSoN Inspector I

Using the Inspector Tool I

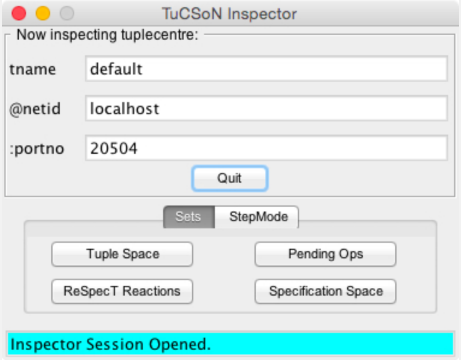
If you launched it without specifying the full name of the target tuple centre to inspect, do it from in the GUI



Using TuCSoN Inspector II

Using the Inspector Tool II

If you launched it giving the full name of the target tuple centre to inspect, choose what to inspect



Using TuCSoN Inspector III

Monitoring of the coordination space is available through the *Sets* tab:

Sets tab

Tuple Space — the *ordinary* tuples space state

Specification Space — the (ReSpecT) *specification* tuples space state

Pending Ops — the *pending* TuCSoN operations set, that is, the set of the operations already issued but currently suspended (waiting for completion)

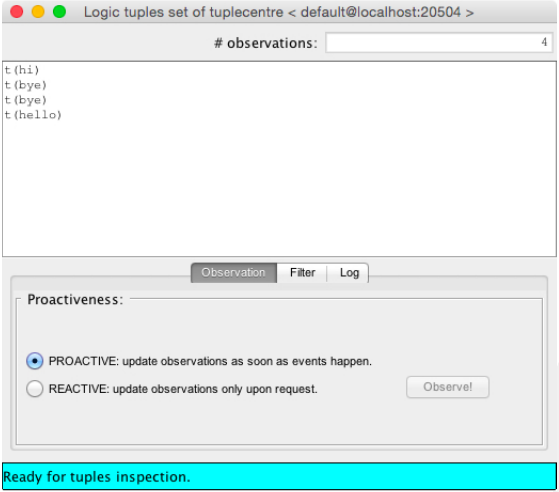
ReSpecT Reactions — the *triggered* (ReSpecT) reactions set, that is, the set of those specification tuples triggered by the TuCSoN operations issued

Using TuCSoN Inspector IV

Tuple Space view

- *Proactively* observe the space state, thus getting any change of state, or *reactively* do so, thus getting updates only when requested
- Filter displayed tuples according to a given template — *Filter* tab
- Log (filtered) observations on a given log file — *Log* tab

Using TuCSoN Inspector V

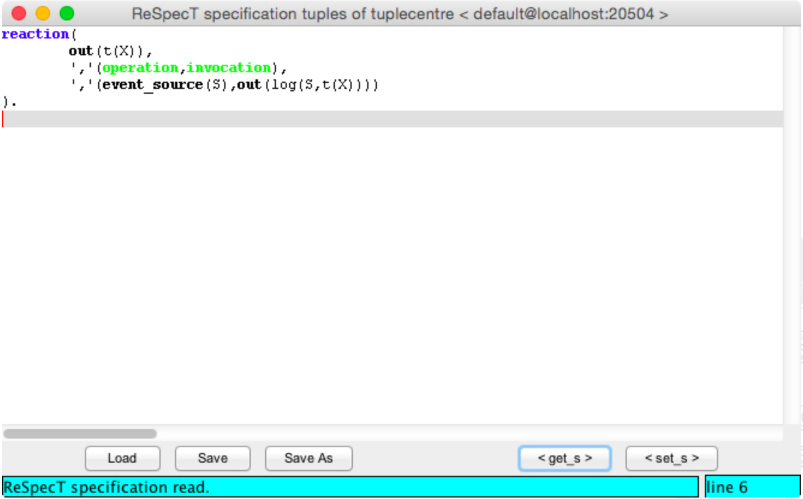


Using TuCSoN Inspector VI

Specification Space view

- Load a ReSpecT specification from a file...
- ...Edit & Set it to the current tuple centre
- Get the ReSpecT specification from the current tuple centre...
- ...Save it to a given file (or to the default one named `default.rsp`)

Using TuCSoN Inspector VII

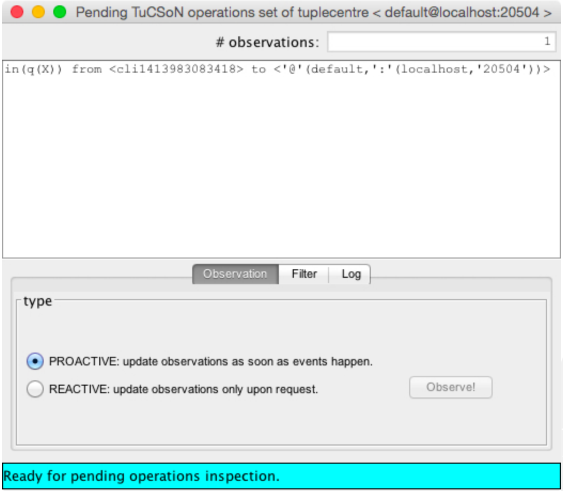


Using TuCSoN Inspector VIII

Pending Ops view

- *Proactively* or reactively observe pending TuCSoN operations
- Filter displayed TuCSoN operations according to a given template — *Filter* tab
- Log (filtered) observations on a given log file — *Log* tab

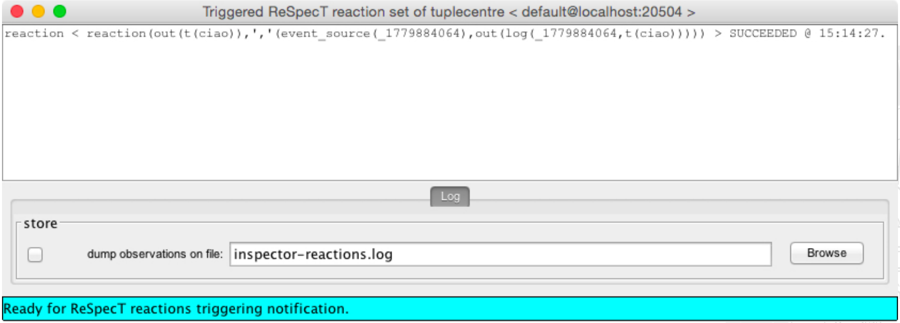
Using TuCSoN Inspector IX



Using TuCSoN Inspector X

ReSpecT *Reactions* view

In the ReSpecT *Reactions* view you are notified on the outcome of any ReSpecT reaction triggered in the observed tuple centre, and can log such notifications on a given log file



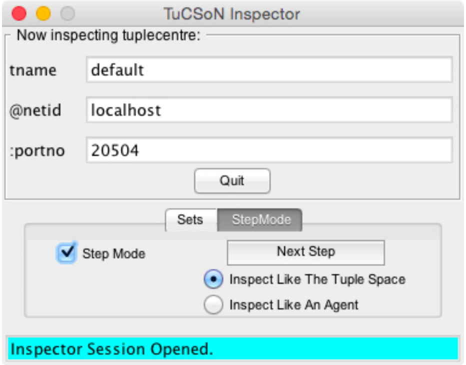
Using TuCSoN Inspector XI

Interaction with the ReSpecT VM is available through the *StepMode* tab:

StepMode tab

- The tuple centre working cycle is paused
- No further processing of incoming events, pending queries, triggering reactions is done
- ReSpecT VM performs transitions between its states only upon pressing of the *Next Step* button
  - ! *one* ReSpecT event is processed at *each* step

Using TuCSoN Inspector XII



Using TuCSoN Inspector XIII

- The radio buttons under the Next Step button let the inspector choose *which point of view* to keep while inspecting the tuple centre:
    - while adopting the tuple centre standpoint, all the Inspector views are updated *at each state transition* — e.g., in the middle of a reaction execution
    - while adopting the agents standpoint, Inspector views are updated *only when a complete VM cycle has been done* — that is, from “idle” state back into it

Outline

Use TuCSoN as a Library

To enable a Java application to use the TuCSoN technology:

1. build a `TucsonAgentId` to be identified by the TuCSoN system
2. get a TuCSoN ACC to enable interaction with the TuCSoN system
3. define the tuple centre target of your coordination operations
4. build a tuple using the communication language
5. perform the coordination operation using a coordination primitive
6. check requested operation success
7. get requested operation result

Let's try!

Launch Java class `HelloWorld` in package `alice.tucson.examples.helloWorld` within TuCSoN distribution and check out code comments

Use TuCSoN as a Framework

To create a TuCSoN agent, do the following:

1. extend `alice.tucson.api.TucsonAgent` base class
2. choose one of the given constructors
3. override the `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()`

Let's try!

Launch Java class `HelloWorldAgent` in package `alice.tucson.examples.helloWorld` within TuCSoN distribution and check out code comments

API Walkthrough I

Package `alice.tucson.api`

Most of the API is made available through package `alice.tucson.api`

- `TucsonAgentId` — exposes methods to build a TuCSoN agent ID, and to access its fields. Required to obtain an ACC

  `getAgentName(): String` — to get the local agent name

- `TucsonMetaACC` — provides TuCSoN agents with a *meta-ACC*, necessary to acquire an ACC, which in turn is mandatory to interact with a TuCSoN tuple centre

  `getAdminContext(TucsonAgentId, String, int, String, String): AdminACC` — to get an *administrator* ACC from the (specified) TuCSoN node

  `getNegotiationContext(TucsonAgentId, String, int): NegotiationACC` — to get a *negotiation* ACC from the (specified) TuCSoN node

API Walkthrough II

- `TucsonTupleCentreId` — exposes methods to build a TuCSoN tuple centre ID, and to access its fields. Required to perform TuCSoN operations on the ACC

  `getName(): String` — to get the tuple centre local name
  `getNode(): String` — to get the tuple centre host's (TuCSoN node) IP number
  `getPort(): int` — to get the tuple centre host's (TuCSoN node) TCP port number

- `ITucsonOperation` — exposes methods to access the result of a TuCSoN operation

  `isResultSuccess(): boolean` — to check operation success
  `getLogicTupleResult(): LogicTuple` — to get operation result

API Walkthrough III

- `AbstractTucsonAgent` — base abstract class for user-defined TuCSoN agents. Automatically builds the `TucsonAgentId` and gets an the `EnhancedACC`

  `main(): void` — to be overridden with the agent's business logic
  `getContext(): EnhancedACC` — to get the ACC for the user-defined agent
  `go(): void` — to start `main` execution of the user-defined agent

- `AbstractSpawnActivity` — base abstract class for user-defined *activities* to be spawned by a `spawn` operation — more on this in slide 176. Provides a simplified syntax for TuCSoN operation invocations

  `doActivity(): void` — to override with the spawned activity's business logic
  `out(LogicTuple): LogicTuple` — `out` TuCSoN operation *bound* to *local* tuple centre
  ... ...

API Walkthrough IV

- `Tucson2PLibrary` — allows tuProlog agents to access the TuCSoN platform by exposing methods to manage ACC, and to invoke TuCSoN operations

  `acquire_acc_1(Struct): boolean` — to get an ACC for the tuProlog agent
  `out_2(Term, Term): boolean` — out TuCSoN operation
  `...` ...

Furthermore. . .

Package `alice.tucson.api` contains also *all the ACC* provided by the TuCSoN platform, among which `EnhancedACC` — those depicted in slide 52

API Walkthrough V

Package `alice.logictuple`

The part of TuCSoN API concerned with managing tuples is made available through package `alice.logictuple`

- `LogicTuple` — exposes methods to build a TuCSoN *tuple/template* and to get its arguments

  `parse(String): LogicTuple` — to encode a given string into a TuCSoN tuple/template

  `getName(): String` — to get the functor name of the tuple

  `getArg(int): TupleArgument` — to get the tuple argument at given position

API Walkthrough VI

- `TupleArgument` — represents TuCSoN tuples arguments (tuProlog *terms*), and provides the means to access them

  `parse(String): TupleArgument` — to encode the given string into a tuProlog tuple argument

  `getArg(int): TupleArgument` — to get the tuple argument at given position

  `isVar(): boolean` — to test if the tuple argument is a tuProlog `Var`

  `intValue(): int` — to get the `int` value of the tuple argument — if admissible

  `...` ...

API Walkthrough VII

Package `alice.tucson.service`

The API to programatically boot & kill a TuCSoN service is provided by class `TucsonNodeService` in package `alice.tucson.service`

- constructors to set-up the TuCSoN service
- methods to install, shutdown, and test installation of the TuCSoN service

  ```
  install(): void
  shutdown(): void
  isInstalled(String, int, int): boolean
  ```

- entry point to launch a TuCSoN node from the command line

The API in Practice

Package `alice.tucson.examples.*`

- `.helloWorld` package
- `.messagePassing` package
- `.rpc` package
- `.masterWorkers` package

Outline

Outline

Meta-Coordination Language

- TuCSoN meta-coordination operations are built out of meta-coordination primitives and of ReSpecT specification languages
  - the specification language
  - the specification template language
- In TuCSoN, both the specification and the specification template languages are *logic-based*, and defined by ReSpecT
  - *any* ReSpecT reaction is an *admissible* TuCSoN specification tuple. . .
  - . . . and an *admissible* TuCSoN specification template

  ⇒ the two languages coincide

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

  invocation phase — the request of the agent reaches the tuple centre, decorated with information about the invocation

  completion phase — the response of the tuple centre goes back to the agent, including information about operation execution outcome

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

tcid ? *op*_s

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

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

Meta-Coordination Primitives

The TuCSoN meta-coordination language provides the following meta-coordination primitives to build meta-coordination operations:

- `out_s` to put a specification tuple in the specification space of the target tuple centre (tc)
- `rd_s`, `rdp_s` to read a specification tuple matching a given specification template from the target tc
- `in_s`, `inp_s` to withdraw a specification tuple matching a given specification template from the target tc
- `no_s`, `nop_s` to check absence of a specification tuple matching a given specification template in the target tc
- `get_s` to read all the specification tuples in the target tc
- `set_s` to overwrite the set of specification tuples in the target tc

Reaction Specification Tuples (ReSpecT)

As a behaviour specification language, ReSpecT

- ✓ enables definition of *computations* within a tuple centre (reactions)
  - ! sequences of logic predicates and functions, and ReSpecT primitives, executing (as a whole) atomically and transactionally, with a *global success/failure semantics*
- ✓ enables association of reactions to events occurring in a tuple centre
  - ⇒ given a ReSpecT event Ev, a specification tuple `reaction(`E`,`G`,`R`)` associates a reaction $R\theta$ to Ev *if and only if* $\theta = \mathtt{mgu}(E,Ev)$[6] *and* guard predicate G evaluates to `true` [Omi07]

ReSpecT twofold interpretation

So, ReSpecT has both a *declarative* and a *procedural* part

[6] Where `mgu` is the *most general unifier*, as defined in logic programming

Reactions Semantics

- ✓ A ReSpecT reaction succeeds *if and only if all* its reaction goals succeed, and fails otherwise
- ✓ Each reaction is executed with a transactional semantics
  - ⇒ hence, a failed reaction has *no effect* on the state of the tuple centre
- ✓ Sequences of reactions are executed sequentially, according to a non-deterministic order,
- ✓ (Sequences of) reactions are executed atomically, that is, before serving other ReSpecT *events*
  - ⇒ thus, agents are *transparent* both to *reactions chaining*, and to multiple reactions triggering for the same event

ReSpecT VM 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 VM Execution Cycle II

When the tuple centre is idle, that is, no reaction is executing

Triggering

1. the first event ϵ in *InQ*, according to a *FIFO policy*, is moved to the multiset *Op* of the *pending* requests
2. consequently, reactions to the invocation phase of ϵ are *triggered* by adding them to the multiset *Re* of the triggered reactions
3. then, those reactions whose guard predicates evaluate to `true` are *scheduled* for execution, while others are removed from *Re*
4. finally, reactions still in *Re* are *executed* — sequentially, non-deterministic order

ReSpecT VM Execution Cycle III

Chaining & linking

Each reaction may trigger

- further reactions, orderly added to *Re*
- output events, representing link invocations, which are
  1. added to the multiset *Out* of the *outgoing events*
  2. then moved to the *output queue* (*OutQ*) of the tuple centre — *if and only if* reaction execution is successful

ReSpecT VM Execution Cycle IV

Completion

Only when *Re* is *empty*

1. pending requests in *Op* are (possibly) executed
2. operation/link completions are sent back to invokers

[!] Further reactions may be raised accordingly, associated to the completion phase of the original invocation, and executed with the *same semantics* specified above for the invocation phase.

Familiarise with ReSpecT I

Let's try!

1. Launch a TuCSoN node on default port
2. Launch TuCSoN Inspector tool — that is, class `InspectorGUI` in package `alice.tucson.introspection.tools`
3. Inspect the `bagoftask` tuple centre on the node
4. Activate "Step Mode" in the "StepMode" tab
5. Launch class `BagOfTaskTest` in package `alice.tucson.examples.respect.bagOfTask`
6. Click "Next Step" on the Inspector to proceed through ReSpecT VM's execution cycle

Malleability of ReSpecT Tuple Centres

- The behaviour of a ReSpecT tuple centre is thus defined by the ReSpecT tuples in the specification space
  - ✓ and it can be adapted by changing its ReSpecT specification *at run-time*

⇒ ReSpecT tuple centres are thus malleable

- ✓ by engineers, via TuCSoN tools — CLI & Inspector
- ✓ by processes, via `in_s` & `out_s` primitives
  - `in` & `out` for the tuple space; `in_s` & `out_s` for the specification space
  - through either an agent coordination primitive, or another tuple centre link operation

Familiarise with ReSpecT II

Let's try!

- Look into the code of the `Master` class in package `alice.tucson.examples.respect.bagOfTask` to see
  - ✓ usage of meta-coordination operations (therefore, primitives) at run-time
  - ✓ structure of ReSpecT specification tuples
- If you wish, change / add / remove invocations and see what happens :)

Linkability of ReSpecT Tuple Centres

- Every tuple centre coordination primitive is also a ReSpecT primitive for reaction goals ("internal"), and a primitive for linking, too
  - ✓ all primitives could be executed within a ReSpecT reaction
    - as either an internal primitive on the same tuple centre
    - or as a link primitive invoked upon another tuple centre
  - ✓ linking primitives are asynchronous — as agent ones
    - ⇒ so they *do not affect* the transactional semantics of reactions
  - ✓ reactions can handle both primitive invocations & completions
- 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[7]

Hold on

Examples showcasing ReSpecT linkability feature are available later on :)

[7] The TuCSoN infrastructure is used for distributing ReSpecT tuple centres

Outline

ReSpecT Syntax: Basics

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
- Specification tuples are logic tuples of the form `reaction(`E`,`G`,`R`)`
  - ✓ if event *Ev* occurs in the tuple centre
  - ✓ if *Ev* matches event descriptor E such that $\theta = mgu(E,Ev)$
  - ✓ if guard G evaluates to `true`
  - ⇒ reaction $R\theta$ is triggered for execution

ReSpecT Syntax: Behaviour Specification

Reference example

We use `table.rsp` ReSpecT specification in package `alice.tucson.examples.diningPhilos` as our running example to back up description of the ReSpecT language following in next slides

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

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

ReSpecT Syntax: Event Descriptor

$$\langle \mathit{Event} \rangle \quad ::= \quad \langle \mathit{Predicate} \rangle \; (\; \langle \mathit{Tuple} \rangle \;) \; | \; \ldots$$

- The simplest event descriptor $\langle \mathit{Event} \rangle$ is the invocation of a primitive $\langle \mathit{Predicate} \rangle \; (\; \langle \mathit{Tuple} \rangle \;)$
- An event descriptor $\langle \mathit{Event} \rangle$ works as a *event template* for matching with admissible events

ReSpecT Syntax: Admissible Events

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

- 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(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

Notice

Understanding the difference between admissible events $\langle TCEvent \rangle$ and event descriptors $\langle Event \rangle$ is essential to understand the main issues of pervasive systems

- admissible events are how we capture and model relevant events: essentially, our *ontology for events*
- event descriptors are how we write about events: essentially, our *language for events*

Role of Coordination Media

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

! this is an essential point in *any* technology dealing with situated computations!

ReSpecT Syntax: Guards

$$\begin{array}{rcl}
\langle \mathit{Guard} \rangle & ::= & \langle \mathit{GuardPredicate} \rangle \mid \\
 & & (\, \langle \mathit{GuardPredicate} \rangle \, \{ \, , \, \langle \mathit{GuardPredicate} \rangle \} \,) \\
\langle \mathit{GuardPredicate} \rangle & ::= & \texttt{request} \mid \texttt{response} \mid \texttt{success} \mid \texttt{failure} \\
 & & \texttt{endo} \mid \texttt{exo} \mid \texttt{intra} \mid \texttt{inter} \\
 & & \texttt{from_agent} \mid \texttt{to_agent} \mid \texttt{from_tc} \mid \texttt{to_tc} \mid \ldots
\end{array}$$

- A triggered reaction is actually executed *only if* its guard evaluates to `true`
- All guard predicates are *ground* ones
- Guard predicates concern properties of the triggering event, so they enable *fine-grained* selection of events

ReSpecT Syntax: Reactions Body I

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

ReSpecT Syntax: Reactions Body 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 atomically and transactionally, with an overall success / failure semantics

- Tuple centre predicates are uniformly used for agent invocations, internal operations, and link invocations
- Similar predicates are used for changing the specification state
  - *pred*_s invocations affect the specification state, and can be used within reactions, also as links

ReSpecT Syntax: Observation Predicates I

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

- `event` & `start` refer to *direct* and *prime* cause, respectively
- `current` refers to what is currently happening, whenever this means something useful — e.g., `current_predicate` is not so useful =)

ReSpecT Syntax: Observation Predicates II

Any combination of the following is admissible in ReSpecT

⟨*EventView*⟩ — allow 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*⟩ — allow 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

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...

...later on we will fix the issue, using ReSpecT :)

Outline

TuCSoN API for ReSpecT

Uniform w.r.t. TuCSoN API to access the ordinary tuple space:

1. build a `TucsonAgentId`
2. get a TuCSoN ACC enabling access to ReSpecT specification space
3. define the tuple centre target of your meta-coordination operations
4. build a specification tuple using the meta-communication language
    - `LogicTuple event = LogicTuple.parse("out(t(X))");`
    - `LogicTuple guards = LogicTuple.parse("(completion, success)");`
    - `LogicTuple reaction = LogicTuple.parse("(in(t(X)), out(tt(X)))");`
5. perform the meta-coordination operation using a meta-coordination primitive
    - `ITucsonOperation op = acc.out_s(tcid, event, guards, reaction, null);`
6. check requested operation success
7. get requested operation result

ReSpecT API

Full ReSpecT API

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

API Examples

Let's try!

Check out examples in package `alice.tucson.examples.*`

1. `.respect.bagOfTask`
2. `.diningPhilos`
3. `.distributedDiningPhilos` — linking primitives here

Outline

Outline

Situatedness in the Spatio-Temporal Fabric I

What is Situatedness?

- Situatedness is essentially the property of systems of being *immersed* in their environment

$\Rightarrow$ 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 [ZCF+11]

Situatedness in the Spatio-Temporal Fabric II

- 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 systems 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*?

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?

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

Time-aware Coordination Media I

A time-aware coordination medium should satisfy the following *requirements* [ORV07]:

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*

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

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

- Since 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:

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

✓ Two ReSpecT *guard predicates* are introduced:

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

- along with the obvious alias

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

Time-aware Coordination Media IV

Capturing time events *(req. 4)*

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

✓ The ReSpecT admissible event model is extended to include time events:

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

✓ 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).
```

Time-aware Coordination Media V

Changing coordination policies over time *(req. 5)*

5. A coordination medium should allow coordination policies to be *changed over time*

✓ It is enough to exploit malleabilty of ReSpecT tuple centres

- exploiting the same ⟨*ForgePredicate*⟩s that can be used for dynamically change tuple centre behaviour at run time
- such as `in_s`, `out_s`, ...

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`

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 [Bea10]

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

Requirements

- ✓ spatial situatedness
- ✓ spatial awareness

(of the coordination media)

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 *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 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* — 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 Coordination Medium: Requirements

Situatedness: Requirements

1. Space should intrinsically belong to the *ontology* of the coordination medium, in terms of *position* and *motion*
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

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

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

- Since every ReSpecT VM executes on a physical (networked) machinery, making an expression of 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 ReSpecT (with TuCSoN) *(req. 3)*

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

- ReSpecT tuple centres have *unique* identities within a TuCSoN node
- Any ⟨*CoordOp*⟩ / ⟨*LinkOp*⟩ can refer to a *tuple centre identifier*, relying on the *local* TuCSoN node as the default local (coordination) space
- Adding a *node reference* to a ⟨*CoordOp*⟩ / ⟨*LinkOp*⟩ 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*:

$$
\begin{aligned}
\langle \mathit{ObservationPredicate} \rangle &::= \langle \mathit{EventView} \rangle\_\langle \mathit{EventInformation} \rangle \\
\langle \mathit{EventView} \rangle &::= \texttt{current} \mid \texttt{event} \mid \texttt{start} \\
\langle \mathit{EventInformation} \rangle &::= \ldots \mid \texttt{place(} \langle \mathit{Space{:}Place} \rangle \texttt{)} \mid \ldots
\end{aligned}
$$

✓ Three ReSpecT guard predicates are introduced:

$$
\langle \mathit{GuardPredicate} \rangle ::= \ldots \mid \texttt{at(} \langle \mathit{Space{:}Place} \rangle \texttt{)} \mid \texttt{near(} \langle \mathit{Space{:}Place} \rangle \texttt{,} \langle \mathit{Radius} \rangle \texttt{)} \mid \ldots
$$

ReSpecT Tuple Centres as Space-aware Media IV

Capturing spatial events *(req. 5)*

5. A coordination medium should be able to *capture* spatial events, and to *react* appropriately

✓ ReSpecT admissible event model is extended to include spatial events

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

ReSpecT Tuple Centres as Space-aware Media V

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

✓ Correspondingly, ReSpecT event descriptor is extended, too

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

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

```
reaction(from(@S,?P), Guard, Body).
 reaction(to(@S,?P), Guard, Body).
```

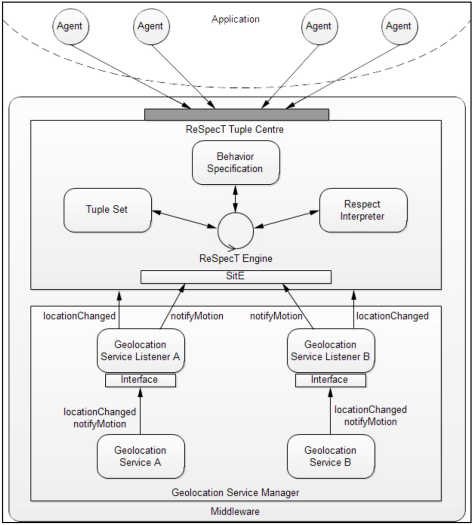
Sorts of Space

It should be noted that the tuple centre position `P` can be specified as either

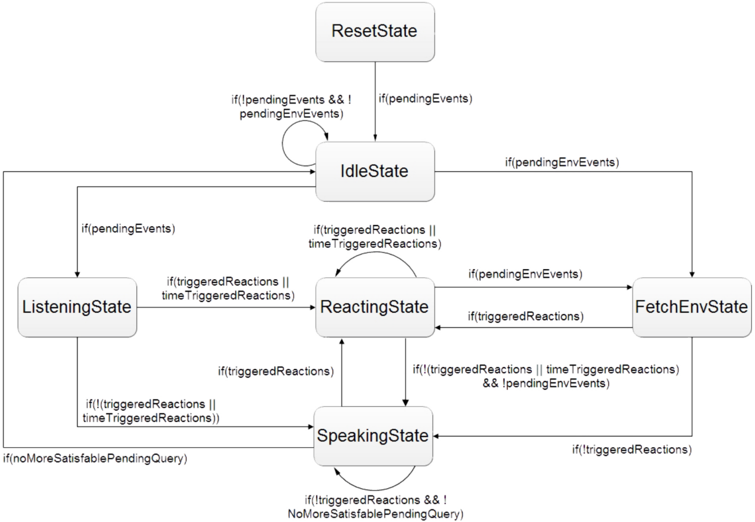
- S = `ph` its absolute physical position
- S = `ip` its IP number
- S = `dns` its domain name
- S = `map` its geographical location — as typically defined by mapping services like Google Maps
- S = `org` its organisational position — that is, a location within an application-defined virtual topology

Space-aware Middleware: TuCSoN on Android I

Space-aware Middleware: TuCSoN on Android II



Space-aware Middleware: TuCSoN on Android III



Outline

Situatedness & Coordination

- Situatedness means, essentially, *strict coupling* with the environment
  - technically, the ability to properly *perceive* and *react* to changes in the environment — possibly, *affecting* it in turn
- One of the most critical issues in distributed systems
  - ! conceptual clash (w.r.t. autonomy) between *pro-activeness* in process behaviour and *reactivity* w.r.t. environment change
- Still a critical issue for artificial intelligence & robotics [Suc87]

What about coordination?

Essentially, situatedness concerns *interaction* between processes and the environment

$\Rightarrow$ thus, situatedness can be conceived as a coordination problem

? how to handle and govern interaction between pro-active processes and an ever-changing environment?

Situating ReSpecT

Situating ReSpecT

- *Situating* the ReSpecT language basically means making ReSpecT capable of *capturing* environment events, and expressing general *MAS-environment interactions* [CO09]

✓ ReSpecT captures, reacts to, and observes general environment events

✓ ReSpecT can explicitly interact with (affect) 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 the environment:

$$\begin{aligned}
\langle \mathit{ObservationPredicate} \rangle &::= \langle \mathit{EventView} \rangle_\langle \mathit{EventInformation} \rangle \\
\langle \mathit{EventView} \rangle &::= \texttt{current} \mid \texttt{event} \mid \texttt{start} \\
\langle \mathit{EventInformation} \rangle &::= \ldots \mid \texttt{env(}\,\langle \mathit{Key} \rangle\,\texttt{,}\,\langle \mathit{Value} \rangle\,\texttt{)}
\end{aligned}$$

✓ Two ReSpecT *guard predicates* are introduced:

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

ReSpecT Tuple Centres as Environment-aware Media II

Capturing general environment events

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

✓ The ReSpecT admissible event model is extended to include environment events

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

ReSpecT Tuple Centres as Environment-aware Media III

Capturing general environment events *(contd.)*

✓ Correspondingly, ReSpecT event descriptor is extended, too

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

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

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

Transducers as Environment Mediators I

- *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 *situation predicate* of the form
    - ✓ ⟨*EResId*⟩ `? get(`⟨*Key*⟩,⟨*Value*⟩`)`
      enabling a tuple centre to *get* properties of environmental resources
    - ✓ ⟨*EResId*⟩ `? set(`⟨*Key*⟩,⟨*Value*⟩`)`
      enabling a tuple centre to *set* properties of environmental resources

Transducers as Environment Mediators II

Specific environment events have to be translated into well-formed ReSpecT tuple centre events

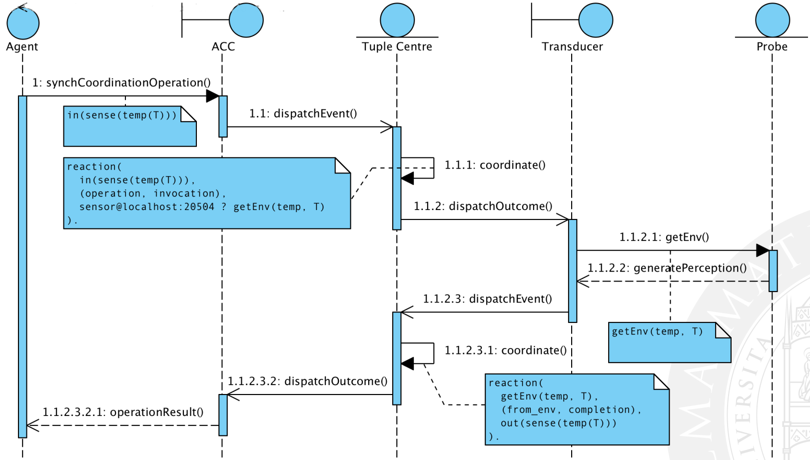
! this should be done at the *infrastructure level*, through a general-purpose schema that could be specialised according to the nature of any specific resource

Transducers

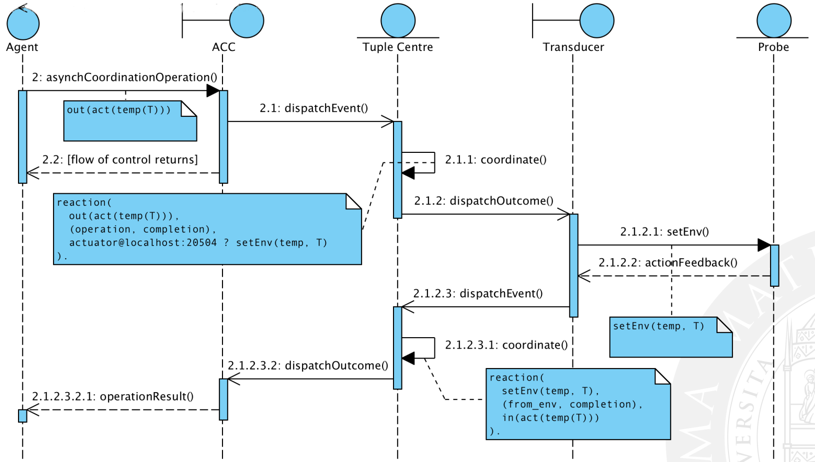
A TuCSoN transducer is a component able to bring environment-generated events to a ReSpecT tuple centre (and back), suitably *translated* according to the general ReSpecT event model

- 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

Sensor / Actuator Transducer Role I



Sensor / Actuator Transducer Role II



Sensor / Actuator Transducer Role III

1. After event dispatching, the tuple centre target of the operation reacts by triggering the ReSpecT reaction in annotation 1.1.1 (2.1.1), which generates a situated event (step 1.1.2 / 2.1.2, respectively) aimed at executing a situation operation (`getEnv(temp, T)` / `getEnv(temp, T)`) on the probe (`sensor` / `actuator`)
2. The transducer associated to the tuple centre and responsible for the target probe *intercepts* such an event and takes care of actually executing the operation on the probe (message 1.1.2.1 / 2.1.2.1)
3. The sensor probe reply (message 1.1.2.2 / 2.1.2.2) generates a sequence of events propagation terminating in the response to the original *coordination operation* issued by the agent (message 1.1.2.3.2.1 / 2.1.2.3.2.1)

Sensor / Actuator Transducer Role IV

Supporting Situatedness

TuCSoN transducers play a central role in supporting *distribution* and uncoupling of agents and probes within the MAS, while TuCSoN tuple centres and the ReSpecT language are fundamental to support both situatedness and *objective* coordination [Sch01, OO03]

Environment Engineering in TuCSoN: Overview I

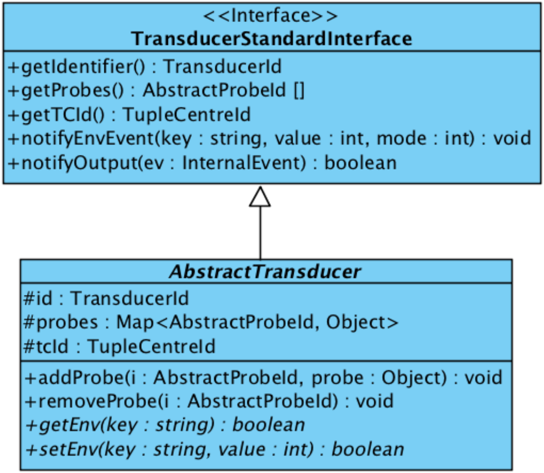
1. Implement probes — sensors and actuators. Typically, this does not require implementing, e.g., software drivers for the probe: designers can simply wrap existing drivers in a Java class implementing the `ISimpleProbe` interface, then interact with TuCSoN transducers
2. Implement transducers associated 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

Environment Engineering in TuCSoN: Overview II

<<Interface>>
ISimpleProbe

+getIdentifier() : AbstractProbeId
+getTransducer() : TransducerId
+setTransducer(tId : TransducerId) : void
+readValue(key : String) : boolean
+writeValue(key : string, value : int) : boolean

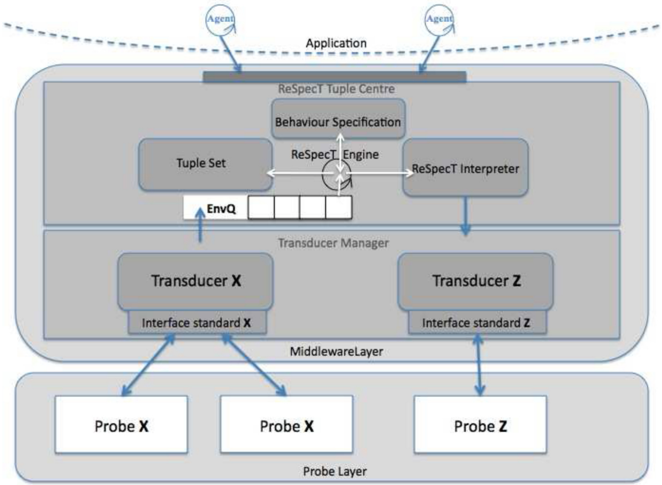
Environment Engineering in TuCSoN: Overview III



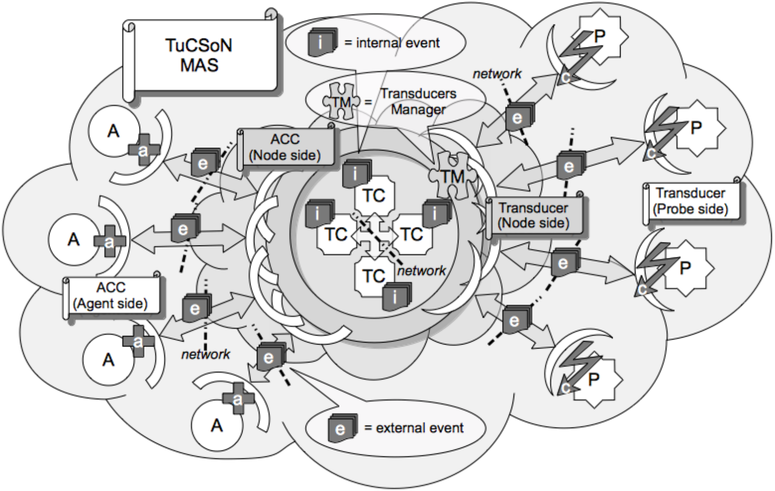
Environment Engineering in TuCSoN: Overview IV

```
<<enumeration>>
TransducersManager
-----------------------------------------------------------------------------
-probesToTransducersMap : Map<TransducerId, List<AbstractProbeId>>
-transducersList : Map<TransducerId, AbstractTransducer>
-transducersToTupleCentresMap : Map<TupleCentreId, List<TransducerId>>
-----------------------------------------------------------------------------
+createTransducer(className : string, id : TransducerId, tcId : TupleCentreId, probeId : AbstractProbeId) : boolean
+addProbe(id : AbstractProbeId, tId : TransducerId, probe : ISimpleProbe) : boolean
+removeProbe(probe : AbstractProbeId) : boolean
+getTransducer(tId : string) : TransducerStandardInterface
+stopTransducer(id : TransducerId) : void
```

TuCSoN-ReSpecT Situated Architecture I



TuCSoN-ReSpecT Situated Architecture II



Example & Further References

Let's try

Check out example `Thermostat` in package

`alice.tucson.examples.situatedness`

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

Outline

Outline

Don't Care Non-determinism

A foremost feature of computational models for *open*, *adaptive* and *self-** systems is non-determinism.

The LINDA approach

LINDA features *don't know* non-determinism handled with a *don't care* approach:

- **don't know** which tuple among the matching ones is retrieved by a getter operation (`in`, `rd`) can be neither specified nor predicted
- **don't care** nonetheless, the coordinated system is designed so as to keep on working whichever is the matching tuple returned

This is not the case, however, in many of today adaptive and self-organising systems, where processes may need to implement stochastic behaviours like *"most of the time do this"*.

Uniform Primitives: Definition I

- Uniform coordination primitives [GVCO07] are required to inject probability within coordination, thus to obtain *stochastic behaviours* in coordinated systems [Omi12b]
- Uniform primitives *replace* the don't know non-determinism of LINDA-like primitives with a uniform probabilistic non-determinism
  - ⇒ so, the tuple returned by a uniform primitive is still chosen non-deterministically among all the tuples matching the template
  - ⇒ however, the choice is now performed with a *uniform distribution*

Uniform Primitives: Definition II

Situation & prediction

Uniform primitives replace don't know non-determinism with *probabilistic non-determinism* to

- ✓ situate a primitive invocation in space
  - ⇒ uniform getter primitives return matching tuples based on the other tuples in the space—so, their behaviour is *context aware*
- ✓ predict its behaviour in time
  - ⇒ sequences of uniform getter operations tend to globally exhibit a *uniform distribution* over time

! Uniform primitives are the *"basic mechanisms enabling self-organising coordination"*, that is, a minimal construct able (alone) to impact the observable properties of a coordinated system.

Uniform Primitives: Definition III

The TuCSoN coordination language provides the following 6 uniform coordination primitives

- `urd`, `uin`
- `urdp`, `uinp`
- `uno`, `unop`

Uniform Primitives: Usage I

- How do you roll a dice in Java?
  (just think about it)
- How do you roll a dice in LINDA?
  (run the example code as it is)
- How do you roll a dice with uniform primitives?
  (run the example code toggling comments on lines 146-147)

Let's try!

Check out `DicePlayer` class in package `alice.tucson.examples.uniform.dice`

Uniform Primitives: Usage II

- How do you guarantee *fairness* of tasks distribution in Java?
  (just think about it)
- How do you guarantee *fairness* of tasks distribution in LINDA?
  (run the example code as it is)
- How do guarantee *fairness* of tasks distribution with uniform primitives?
  (run the example code toggling comments on lines 115-116)

Let's try!

Check out "Load Balancing" example in package `alice.tucson.examples.uniform.loadBalancing` — use TuCSoN Inspector tool

Uniform Primitives: Usage III

- How do you inject a *controlled bias* in a random-based behaviour in Java?
  (just think about it)
- How do you do so in LINDA?
  (just think about it)
- How do you do so with uniform primitives?
  (run the example code)

Let's try!

Check out `LaunchSwarmsScenario` class in package `alice.tucson.examples.uniform.swarms.launchers`

Further References

Paper

http://scs.org/documents/Simulation/2_MarianiOmicini.pdf

Outline

Outline

Bulk Primitives

- Bulk coordination primitives provide efficiency gains when dealing with multiple tuples, allowing usage of a *single coordination operation* to return the *whole set* of tuples matching a given template [Row96]
  - ! In case no matching tuples are found, they successfully complete anyway, returning an *empty list* of tuples
- The TuCSoN coordination language provides the following 4 bulk coordination primitives:

  `out_all` puts the given (Prolog) list of tuples in the target tuple centre

  `rd_all` reads all the tuples matching the given template from the target tuple centre

  `in_all` withdraws all the tuples matching the given template from the target tuple centre

  `no_all` checks absence of tuples matching the given template in the target tuple centre

Bulk Primitives: Example

Let's try!

Check out "Master-Workers" example in package `alice.tucson.examples.masterWorkers.bulk`

The spawn Primitive

To delegate computational activities related to coordination to the coordination medium itself, TuCSoN provides the `spawn` primitive — similar to LINDA `eval`

Semantics

- `spawn` activates a parallel computational activity to be carried out asynchronously w.r.t. the caller
- Execution of `spawn` is local to the tuple centre where it is invoked, and so are its results
  - correspondingly, the code implementing the spawned computation must be *locally available* — no code mobility
  - the spawned computation can execute (a subset of) TuCSoN coordination primitives *only locally* — no remote operations

spawn Primitive: Syntax I

General syntax

- spawn has basically[a] two parameters:

  activity — a ground tuple indicating either
  - the tuProlog theory implementing the activity, along with the goal to trigger resolution—e.g., `solve('path/to/Prolog/Theory.pl', goal)`
  - the Java class implementing the activity—e.g., `exec('list.of.packages.Class.class')`

  tuple centre — a ground tuple identifying the tuple centre in charge of spawn execution—thus, where the activity will take place
- If using tuProlog API, this suffices...

[a] See next slide :)

spawn Primitive: Syntax II

Java-specific syntax

- . . . if using Java API, a third parameter is instead necessary, which is either

  listener — the listener object `TucsonOperationCompletionListener` to notify upon `spawn` completion—in case of an asynchronous invocation

  timeout — the long value determining the maximum waiting time for completion (in milliseconds)—in case of a synchronous call

- In either case, `spawn` execution is still a separate, parallel computation

spawn Primitive: Example

Let's try!

Check out "Spawned Workers" example in package

`alice.tucson.examples.spawnedWorkers`

Outline

Asynchronous Operation Invocation I

- Coordination operations may be invoked in two modes

  synchronous — blocking *the caller agent* whenever the invoked operation gets suspended

  asynchronous — *preserving* agents' own autonomy, by *decoupling* the agent control flow from the coordination operation control flow

  ✓ Asynchronous mode is supported by the `AsynchOpsHelper` TuCSoN component in package `alice.tucson.asynchSupport`, which then keeps track of *pending* and *completed* operations on agents' behalf

Asynchronous Operation Invocation II

The API exposed by `AsynchOpsHelper` consists of

`enqueue(AbstractTucsonAction,TucsonOperationCompletionListener): boolean` — adds an operation to the queue of *pending operations*, given the listener component to notify upon its completion

`getPendingOps(): SearchableOpsQueue` — gets the queue of pending operations, that is, a *thread-safe* queue providing a `getMatchingOps(...)` method to *filter* on operations *type* — e.g., `in`, `rd`, etc.

`getCompletedOps(): CompletedOpsQueue` — gets the queue of *completed operations*, that is, a *thread-safe* queue providing methods to *filter* on operations *features* (type, *outcome*) — e.g., successful operations, failed operations

Asynchronous Operation Invocation III

`shutdownGracefully(): void` — requests *soft shutdown* of the helper, that is, shutdown *waits* for pending operations to complete

`shutdownNow(): void` — requests *hard shutdown* of the helper, that is, shutdown happens as soon as the currently executing operation completes — other pending operations are *discarded*

Further Reference

Details can be found in "Asynchronous Operation Invocation in TuCSoN" how-to at `http://apice.unibo.it/xwiki/bin/view/TuCSoN/Documents`

Let's try!

Check out example `PrimeCalculationLauncher` in package `alice.tucson.examples.asynchAPI`

Persistency & Recovery I

- ✓ TuCSoN supports persistency of both the ordinary tuple space and the specification tuple space
  - ⇒ this means it is possible to move the content of a tuple centre from *volatile* memory to *persistent* storage
- To do so, an XML file is created upon request, storing a *snapshot* of the tuple centre content – "frozen" at the exact moment when persistency is enabled – as well as all the *updates* occurring afterwards — until persistency is disabled

Persistency & Recovery II

- The XML file is created within the `persistent/` folder *in the directory where* TuCSoN *has been installed*
- The XML file is named according to the following scheme `tc_`*`tcname`*`_at_`*`netid`*`_at_`*`portno`*`_`*`yyyy-mm-dd`*`_`*`hh.mm.ss`*, where
  - *`tcname`* is the name of the tuple centre made persistent
  - *`netid`* is the IP address of the TuCSoN node hosting the tuple centre made persistent
  - *`portno`* is the TCP port number of the TuCSoN node hosting the tuple centre made persistent
  - *`yyyy-mm-dd`* is the "year-month-day" date when the persistency file has been created
  - *`hh.mm.ss`* is the "hours.minutes.seconds" time when the persistency file has been created

Persistency & Recovery III

- Within the persistency file, persistent information is encoded in XML as follows:
    - first line is the XML header, declaring XML version, encoding, etc.
    - root element is the `<persistency>` node, with no attributes
    - its first children is node `<snapshot>`, storing the content of the tuple centre when persistency was enabled, with attributes
        - `tc`, a String storing the id of the tuple centre persistency refers to
        - `time`, a String storing the timestamp when the snapshot was last updated (in the same format as previous slide)
    - its second children is node `<updates>`, storing the updates occurred afterwards, with attributes
        - `time`, a String storing the timestamp when the last update was recorder (in the same format as previous slide)

Persistency & Recovery IV

- Node `<snapshot>` has three children nodes:
  - `<tuples>`, storing the ordinary tuples between children nodes `<tuple> </tuple>`, with no attributes
  - `<specTuples>`, storing the specification tuples between children nodes `<specTuple> </specTuple>`, with no attributes
  - `<predicates>`, storing the Prolog predicates (supporting specification tuples) between children nodes `<predicate> </predicate>`, with no attributes
- Node `<updates>` has only one type of children node, `<update>`, storing the ordinary tuple, specification tuple or predicate the update refers to, with attributes to distinguish the *kind of update* recorded (all Strings):
  - `action`, recording if the update is an `addition`, `deletion` or a `clean` (removing all the "subjects" of the action)
  - `subject`, recording if the update refers to a `tuple`, a `specTuple` or a `predicate`

Persistency & Recovery V

tc_default_at_localhost_at_20504_2014-10-21_16.09.36.xml

File Path ▾ : ~/Documents/tucson-space/TuCSoN/persistent/tc_default_at_localhost_at_20504_2014-10-21_16.09.36.xml

tc_default_at_localhost_at_20504_2014-10-21_16.09.36.xml (no symbol selected)

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<persistency>
    <snapshot tc="'@'(default,':'(localhost,20504))" time="2014-10-21_16.09.36">
        <tuples> … </tuples>
        <specTuples> … </specTuples>
        <predicates> … </predicates>
    </snapshot>
    <updates time="2014-10-21_16.09.50">
        <update action="addition" subject="tuple">is_persistent(default)</update>
        <update action="addition" subject="tuple">ttt(2000)</update>
        <update action="addition" subject="tuple">ttt(2001)</update>
        <update action="addition" subject="tuple">ttt(2002)</update>
        <update action="addition" subject="tuple">ttt(2003)</update>
        <update action="addition" subject="tuple">ttt(2004)</update>
        <update action="addition" subject="tuple">ttt(2005)</update>
        <update action="addition" subject="tuple">ttt(2006)</update>
        <update action="addition" subject="tuple">ttt(2007)</update>
        <update action="addition" subject="tuple">ttt(2008)</update>
        <update action="addition" subject="tuple">ttt(2009)</update>
        <update action="addition" subject="tuple">ttt(2010)</update>
        <update action="addition" subject="tuple">ttt(2011)</update>
        <update action="addition" subject="tuple">ttt(2012)</update>
        <update action="addition" subject="tuple">ttt(2013)</update>
        <update action="addition" subject="tuple">ttt(2014)</update>
        <update action="addition" subject="tuple">ttt(2015)</update>
        <update action="addition" subject="tuple">ttt(2016)</update>
        <update action="addition" subject="tuple">ttt(2017)</update>
        <update action="addition" subject="tuple">ttt(2018)</update>
        <update action="addition" subject="tuple">ttt(2019)</update>
        <update action="addition" subject="tuple">ttt(2020)</update>
        <update action="addition" subject="tuple">ttt(2021)</update>
        <update action="addition" subject="tuple">ttt(2022)</update>
        <update action="addition" subject="tuple">ttt(2023)</update>
        <update action="addition" subject="tuple">ttt(2024)</update>
        <update action="addition" subject="tuple">ttt(2025)</update>
        <update action="addition" subject="tuple">ttt(2026)</update>
        <update action="addition" subject="tuple">ttt(2027)</update>
        <update action="addition" subject="tuple">ttt(2028)</update>
        <update action="addition" subject="tuple">ttt(2029)</update>
        <update action="addition" subject="tuple">ttt(2030)</update>
    </updates>
</persistency>
```

Line 2019 Col 9 | XML | Unicode (UTF-8) | Unix (LF) | Last saved: 22/10/14 11:44:09 | 73.035 / 8.342 / 2.050

Persistency & Recovery VI

- The purpose of TuCSoN persistency feature is that of supporting a basic level of fault-tolerance
  - ⇒ in fact, once the content of a tuple centre is on persistent storage, it can be retrieved anytime and restored
  - ✓ thus, in case of, e.g., a TuCSoN node crashes, it is possible to restart it and recover the content of the persistent tuple centres it hosted
- ✓ Recovery of a persistent tuple centre is *automatic* in TuCSoN
  - whenever a TuCSoN node is installed in a directory, on boot it seeks such directory for the `persistent/` folder and *recovers all the tuple centres found*
- Whenever recovering a persistent tuple centre from its XML file, persistency is re-enabled on that tuple centre as soon as the recovery process ends

Persistency & Recovery VII

- Enabling/disabling persistency is as simple as putting/removing a well-defined tuple in the special TuCSoN tuple centre called `'$ORG'`: `cmd(enable_persistency([tcid]))`, where *tcid* is the id of the tuple centre whose persistency feature should be enabled
- As soon as persistency is enabled, the persistency XML file is created and the `<snapshot>` node written; then, the `<updates>` node opened, ready to record updates
- As soon as persistency is disabled, the `<updates>` node is closed and timestamped; then, the persistency file is closed and timestamped too
- Testing if a tuple centre is persistent is as simple as testing it for presence of tuple `is_persistent`, which is automatically added as soon as persistency is enabled and automatically removed when disabled

Persistency & Recovery VIII

Let's try!

Check out example `PersistencyTester` in package `alice.tucson.examples.persistency`

RBAC in TuCSoN

- Role-Based Access Control (RBAC) models[8] integrate *organisation* and *security*, by assigning roles to processes, and by ruling the distributed access to *resources*

✓ TuCSoN implements *RBAC-MAS* [ORV05b], a version of RBAC where organisation and security issues are handled in a uniform way as coordination issues

  ⇒ a special tuple centre (called `$ORG`) contains the *dynamic* rules of RBAC in TuCSoN

[8] `http://csrc.nist.gov/groups/SNS/rbac/`

RBAC API I

- Interface `RBACStructure`
  - implementation class `TucsonRBACStructure`
  - package `alice.tucson.rbac`

  models a RBAC organisation within TuCSoN
- It includes
  - a set of roles, as instances of class `TucsonRole` (interface `Role`)
  - a set of policies, as instances of class `TucsonPolicy` (interface `Policy`)
  - a set of authorised agents, as instances of class `TucsonAutorisedAgent` (interface `AuthorisedAgent`)

RBAC API II

- Class `TucsonRole` includes, besides its name and description:
  - the *policy* it adheres to
  - the agent class associated to the role, allowing activation of the role only for those agents belonging to such class
- Class `TucsonPolicy` includes, besides its name:
  - a set of permissions, as instances of class `TucsonPermission` (interface `Permission`)
- Class `TucsonPermission`, currently, simply represents the name of a TuCSoN primitive, to model the fact that the associated policy allows agents with the associated role to request TuCSoN operations involving that primitive
- Class `TucsonAutorisedAgent` models a *recognised* TuCSoN agent, that is, an agent who performed a successful *login* into RBAC-TuCSoN; as such, it includes the *agent class* the logged agents belongs to, its (encrypted) *username* and (encrypted) *password*

RBAC API III

Further reference

Other RBAC-related properties belonging to the TuCSoN node – hence to `TucsonNodeService` class – can be configured — see "RBAC in TuCSoN" how-to at `http://apice.unibo.it/xwiki/bin/view/TuCSoN/Documents`

RBAC API IV

- To participate a TuCSoN-RBAC organisation, agents need to
  1. acquire a meta-ACC
  2. *activate* a role to acquire an ACC
- Step 1 involves class `TucsonMetaACC`, within package `alice.tucson.api`:
- Step 2 involves the `NegotiationACC`, which lets TuCSoN “clients” acquire an ACC, by *playing* RBAC roles, enabling *restricted* interaction with TuCSoN coordination services
  - ✓ the released ACC is equipped with a *built-in filter* allowing only admissible operations according to the agent’s role

RBAC API V

`playRole(String, Long): EnhancedACC` — attempts to play the given role

`playRoleWithPermissions(List<String>, Long): EnhancedACC` — attempts to play a role given a set of *desired permissions*. The principle according to which a role is selected is the least privilege: among the roles enabling all desired permissions, the one giving the least permissions is selected—if no suitable role is found, no ACC is released

Let's try!

Check out example `RBACLauncher` in package `alice.tucson.examples.rbac`

Outline

TuCSoN4JADE

Home | Documentation | Support | Download | Developers | News | Related Projects

Third Party Contributions

This page contains JADE-related software developed by various users from the JADE community.

The shown information is that reported by the developers themselves. The JADE team is not responsible for the correctness of this information neither for the legal correctness of the license.

File	Since	Author	License	Description
TuCSoN4JADE	JADE 4.3.2	Stefano Mariani	LGPL	TuCSoN4JADE (T4J for short) is a Java library enabling JADE agents to exploit TuCSoN coordination services wrapped as an ad-hoc API into a JADE kernel service. TuCSoN is a Java-based middleware providing software agents with coordination as a service via programmable logic tuple spaces, called tuple centres. By combining TuCSoN and JADE, T4J aims at providing MAS engineers with a full-featured MAS middleware, enabling them to exploit both dimensions of agent-oriented software engineering (individual, through JADE agents; social, via TuCSoN tuple centres) in a complete and well-balanced way.

TuCSoN4*Jason*

Jason

a Java-based interpreter for an extended version of AgentSpeak

Jason
by Gustave Moreau (1865)

Oil on canvas, 204 x 115.5 cm.
Musée d'Orsay, Paris.
© Photo RMN, Photograph by
Hervé Lewandowski

Home | Description | Documents | Examples | Demos | Teaching | **Projects** | Search

Projects

These are some of the project contributing, extending, or using *Jason*:

- J-Moise allows ***Jason*** agents to use Moise+ organisations (available with the ***Jason*** distribution).
- There is an interface between ***Jason*** and Cartago, available here.
- The ***Jason***-Cartago combination has generated two excellent projects:
  - JaCa-Android
  - JaCa-Web
- Jason-ROS is a modular interface between ***Jason***, CArtAgO, and ROS.
- TuCSoN4Jason is a Java library enabling ***Jason*** agents to exploit TuCSoN coordination services wrapped as ***Jason*** internal actions.
- A framework that can be used to deploy multiple concurrent ***Jason*** agents in complex Second Life simulations is described in this paper.

If you know of other projects using or extending ***Jason***, please let us know.

Download the latest version of Jason!

DOWNLOAD

Read the tutorial for installing Jason as an Eclipse plug-in!

ECLIPSE PLUG-IN

News - 14/03/2016

Context

- In objective coordination, coordination-related concerns are extracted from agents to be embodied within dedicated abstractions offering *coordination as a service* [VO06]
- In subjective coordination instead, coordination issues are directly tackled by individual agents themselves

Objective & Subjective Coordination [OO03]

Objective and subjective coordination thus constitute two *complementary* approaches, *both* essential in MAS design and development [ROD03], hence requiring a suitable integration

Motivation

- Successful integration depends on the *technology level*, that is, on the mechanisms provided by the agent frameworks to be integrated
- In particular, it depends on the model of autonomy promoted by the specific agent platform, and by its relationship with the model of coordination adopted by the specific (objective) coordination framework

Hindering Autonomy

Any integration effort *not* taking into account such two aspects is likely to hinder agent autonomy by (unintentionally) creating *artificial dependencies* between the subjective and the objective stances on coordination

The Issue of Autonomy I

Model of Autonomy

A model defining *(i)* how agents behave as *individual* entities, *(ii)* how they relate to each other as *social* entities, as well as *(iii)* how the two things *coexist*

Model of Coordination

A model defining the semantics of the admissible *interactions* between agents in a MAS, in particular, w.r.t. their effects on the agent autonomy (e.g., *control flow*)

The Issue of Autonomy II

JADE Model of Autonomy

- Behaviours for individual tasks
- Asynchronous messages for subjective coordination
- The "`block()`-then-resume" pattern to reconcile individual and social attitudes

Jason Model of Autonomy

- Plans/intentions for individual tasks
- Asynchronous message passing for subjective coordination
- Intention suspension to reconcile individual and social attitudes

The Issue of Autonomy III

TuCSoN Model of Coordination

By decoupling invocation semantics from the operation semantics, synchronous calls are always consequence of the *agent own deliberation* process

The Issue of Autonomy IV

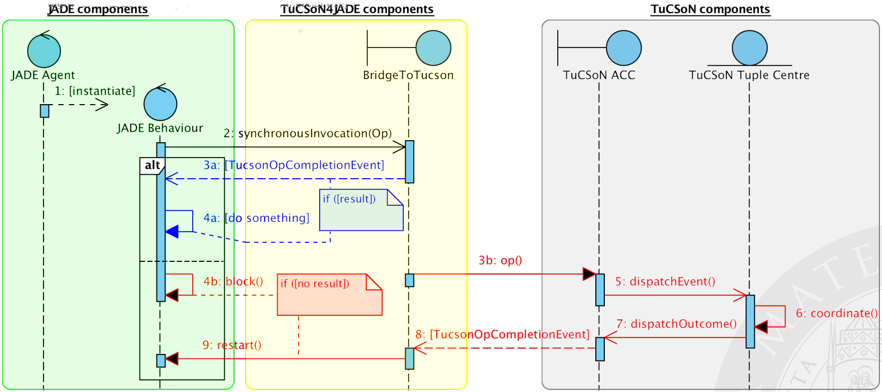


Figure: The "alt"-labelled frame is the equivalent of JADE `blockingReceive()` programming pattern in TuCSoN4JADE.

The Issue of Autonomy V

. . . *Jason*?

The whole approach is the same, obviously the abstractions, mechanisms, and architecture of the solution differs

Further reference

Paper:

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

Codebase:

- `https://bitbucket.org/smariani/tucson4jade`
- `https://bitbucket.org/smariani/tucson4jason`

Outline

Outline

A Novel, Higher Level Language I

```
specification diningPhilos.time_table {

    @chopsReq
    reaction in chops(C1,C2) : invocation, operation {
        out required(C1,C2)
    }

    @cleanChopsReq
    reaction in chops(C1,C2) : completion, operation {
        in required(C1,C2)
    }

    @serveChopsReq
    reaction out required(C1,C2) : internal {
        in chop(C1),
        in chop(C2),
        out chops(C1,C2)
    }

    @chopsRelWait
    reaction out chops(C1,C2) : completion, operation {
        in chops(C1,C2)
    }

    @chopsRelInTime
    reaction out chops(C1,C2) : completion, operation {
        in(used(C1,C2,_)),
        out chop(C1),
        out chop(C2)
    }
```


```
specification diningPhilos.time_table {

    @chopsReq
    reaction in chops(C1,C2) : invocation, operation {
        out required(C1,C2)
    }

    @cleanChopsReq
    reaction in chops(C1,C2) : completion, operation {
        in required(C1,C2)
    }

    @serveChopsReq
    reaction out required(C1,C2) : internal {
        in chop(C1),
        in chop(C2),
        out chops(C1,C2)
    }

    @chopsRelWait
    reaction out chops(C1,C2) : completion, operation {
        in chops(C1,C2)
    }

    @chopsRelInTime
    reaction out chops(C1,C2) : completion, operation {
        in(used(C1,C2,_)),
        out chop(C1),
        out chop(C2)
    }
```

A Novel, Higher Level Language II

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% BEGIN Specification 'time_table'
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
/**
 * @chopsReq
 */
reaction(
    in(chops(C1, C2)),
    (
        invocation,
        operation
    ), (
        out(required(C1, C2))
    )
).

/**
 * @cleanChopsReq
 */
reaction(
    in(chops(C1, C2)),
    (
        completion,
        operation
    ), (
        in(required(C1, C2))
    )
).

/**
 * @serveChopsReq
 */
reaction(
    out(required(C1, C2)),
    (
        internal
    ), (
        in(chop(C1)),
        in(chop(C2)),
        out(chops(C1, C2))
    )
).

/**
 * @chopsRelWait
 */
reaction(
    out(chops(C1, C2)),
    (
        completion,
        operation
    ), (
        in(chops(C1, C2))
    )
).

/**
 * @chopsRelInTime
 */
reaction(
    out(chops(C1, C2)),
    (
        completion,
        operation
    ), (
        in(used(C1, C2, _)),
        out(chop(C1)),
        out(chop(C2))
    )
).

/**
 * @chopsPending1
 */
reaction(
    out(chop(C1)),
    (
        internal
    ), (
        rd(required(C1, C)),
        in(chop(C1)),
        in(chop(C)),
        out(chops(C1, C))
    )
).

/**
 * @chopsPending2
 */
reaction(
    out(chop(C2)),
    (
        internal
    ), (
        rd(required(C, C2)),
        in(chop(C)),
        in(chop(C2)),
        out(chops(C, C2))
    )
).

/**
 * @timeExpired
 */
reaction(
    in(chops(C1, C2)),
    (
        completion,
        operation
    ), (
        current_time(T),
        rd(max_eating_time(Max)),
        T1 is T + Max,
        out(used(C1, C2, T)),
        out_s(
            time(T1),
            (
                internal
            ), (
                in(used(C1, C2, T)),
            out(chop(C1)),
            out(chop(C2))
            )
        )
    )
).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% END Specification 'time_table'
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

A Novel, Higher Level Language III

Features

- Useless variables warning — replaceable by anonymous variable "_"
- Modularity of ReSpecT reactions — `specifications` may include `modules`
- Existence and absence guards — `rd`, `in`, `no` can be used as guards, with and without side effects
- Some syntactic sugar and a more imperative programming style
- A few semantic checks: redundant and conflicting guards errors, same "signature" warning, etc.
- Automatic generation of (editable) ReSpecT specifications

Reference

`https://bitbucket.org/smariani/respectx`

References I

Jacob Beal.
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, 2010. IEEE Computer Society.

Paolo Ciancarini.
Coordination models and languages as software integrators.
ACM Computing Surveys, 28(2):300–302, June 1996.

Matteo Casadei and Andrea Omicini.
Situated tuple centres in ReSpecT.
In Sung Y. Shin, Sascha Ossowski, Ronaldo Menezes, and Mirko Viroli, editors, *24th Annual ACM Symposium on Applied Computing (SAC 2009)*, volume III, pages 1361–1368, Honolulu, Hawai'i, USA, 8–12 March 2009. ACM.

References II

Mehdi Dastani, Farhad Arbab, and Frank S. de Boer.
Coordination and composition in multi-agent systems.
In Frank Dignum, Virginia Dignum, Sven Koenig, Sarit Kraus, Munindar P. Singh, and Michael J. Wooldridge, editors, *4rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005)*, pages 439–446, Utrecht, The Netherlands, 25–29 July 2005. ACM.

Enrico Denti, Andrea Omicini, and Alessandro Ricci.
tuProlog: A light-weight Prolog for Internet applications and infrastructures.
In I.V. Ramakrishnan, editor, *Practical Aspects of Declarative Languages*, volume 1990 of *Lecture Notes in Computer Science*, pages 184–198. Springer Berlin Heidelberg, 2001. 3rd International Symposium (PADL 2001), Las Vegas, NV, USA, 11–12 March 2001. Proceedings.

Martin Fredriksson and Rune Gustavsson.
Online engineering and open computational systems.
In Federico Bergenti, Marie-Pierre Gleizes, and Franco Zambonelli, editors, *Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook*, volume 11 of *Multiagent Systems, Artificial Societies, and Simulated Organization*, pages 377–388. Kluwer Academic Publishers, 2004.

References III

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

David Gelernter.
Generative communication in Linda.
ACM Transactions on Programming Languages and Systems, 7(1):80–112, January 1985.

Dina Q. Goldin, Scott A. Smolka, and Peter Wegner, editors.
Interactive Computation: The New Paradigm.
Springer, September 2006.

Luca Gardelli, Mirko Viroli, Matteo Casadei, and Andrea Omicini.
Designing self-organising MAS environments: The collective sort case.
In Danny Weyns, H. Van Dyke Parunak, and Fabien Michel, editors, *Environments for MultiAgent Systems III*, volume 4389 of *LNAI*, pages 254–271. Springer, May 2007.
3rd International Workshop (E4MAS 2006), Hakodate, Japan, 8 May 2006. Selected Revised and Invited Papers.

John W. Lloyd.
Foundations of Logic Programming.
Springer, 1st edition, 1984.

References IV

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

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

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

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

References V

Andrea Omicini.
Agents writing on walls: Cognitive stigmergy and beyond.
In Fabio Paglieri, Luca Tummolini, Rino Falcone, and Maria Miceli, editors, *The Goals of Cognition. Essays in Honor of Cristiano Castelfranchi*, volume 20 of *Tributes*, chapter 29, pages 543–556. College Publications, London, December 2012.

Andrea Omicini.
Nature-inspired coordination for complex distributed systems.
In *Intelligent Distributed Computing VI*, Studies in Computational Intelligence, Calabria, Italy, 24-26 September 2012. Springer.
6th International Symposium on Intelligent Distributed Computing (IDC 2012). Invited paper.

Andrea Omicini and Sascha Ossowski.
Objective versus subjective coordination in the engineering of agent systems.
In Matthias Klusch, Sonia Bergamaschi, Peter Edwards, and Paolo Petta, editors, *Intelligent Information Agents: An AgentLink Perspective*, volume 2586 of *LNAI: State-of-the-Art Survey*, pages 179–202. Springer, 2003.

References VI

Andrea Omicini, Alessandro Ricci, and Mirko Viroli.
An algebraic approach for modelling organisation, roles and contexts in MAS.
Applicable Algebra in Engineering, Communication and Computing, 16(2-3):151–178, August 2005.
Special Issue: Process Algebras and Multi-Agent Systems.

Andrea Omicini, Alessandro Ricci, and Mirko Viroli.
RBAC for organisation and security in an agent coordination infrastructure.
Electronic Notes in Theoretical Computer Science, 128(5):65–85, 3 May 2005.
2nd International Workshop on Security Issues in Coordination Models, Languages and Systems (SecCo'04), 30 August 2004. Proceedings.

Andrea Omicini, Alessandro Ricci, and Mirko Viroli.
Timed environment for Web agents.
Web Intelligence and Agent Systems, 5(2):161–175, August 2007.

Andrea Omicini and Franco Zambonelli.
Coordination of mobile information agents in TuCSoN.
Internet Research, 8(5):400–413, December 1998.

References VII

Andrea Omicini and Franco Zambonelli.
Coordination for Internet application development.
Autonomous Agents and Multi-Agent Systems, 2(3):251–269, September 1999.
Special Issue: Coordination Mechanisms for Web Agents.

George A. Papadopoulos and Farhad Arbab.
Coordination models and languages.
In Marvin V. Zelkowitz, editor, *The Engineering of Large Systems*, volume 46 of *Advances in Computers*, pages 329–400. Academic Press, 1998.

Alessandro Ricci, Andrea Omicini, and Enrico Denti.
Activity Theory as a framework for MAS coordination.
In Paolo Petta, Robert Tolksdorf, and Franco Zambonelli, editors, *Engineering Societies in the Agents World III*, volume 2577 of *LNCS*, pages 96–110. Springer, April 2003.

Antony Ian Taylor Rowstron.
Bulk Primitives in Linda Run-Time Systems.
PhD thesis, The University of York, 1996.

References VIII

Michael Schumacher.
Objective Coordination in Multi-Agent System Engineering. Design and Implementation, volume 2039 of *LNCS*.
Springer, April 2001.

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

Mirko Viroli and Andrea Omicini.
Coordination as a service.
Fundamenta Informaticae, 73(4):507–534, 2006.
Special Issue: Best papers of FOCLASA 2002.

Peter Wegner and Dina Goldin.
Computation beyond Turing machines.
Communications of the ACM, 46(4):100–102, April 2003.

References IX

Franco Zambonelli, Gabriella Castelli, Laura Ferrari, Marco Mamei, Alberto Rosi, Giovanna Di Marzo Serugendo, Matteo Risoldi, Akla-Esso Tchao, Simon Dobson, Graeme Stevenson, Yuan Ye, Elena Nardini, Andrea Omicini, Sara Montagna, Mirko Viroli, Alois Ferscha, Sascha Maschek, and Bernhard Wally.
Self-aware pervasive service ecosystems.
Procedia Computer Science, 7:197–199, December 2011.
Proceedings of the 2nd European Future Technologies Conference and Exhibition 2011 (FET 11).

Advanced Coordination Techniques

Experiments with TuCSoN and ReSpecT

Stefano Mariani Andrea Omicini
{s.mariani, andrea.omicini}@unibo.it

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

Faculté d'informatique – Université de Namur
Thursday, April 28th, 2016