The TuCSoN Coordination Model & Technology
A Guide

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

Alma Mater Studiorum—Università di Bologna a Cesena

TuCSoN v. 1.12.0.0301
Guide v. 1.3.1
June 15, 2015

Outline of Part I: Basic TuCSoN

Outline of Part II: Advanced TuCSoN

4. Advanced Model
   - Bulk Primitives
   - Coordinative Computation
   - Uniform Primitives
   - Organisation
   - Agent Coordination Contexts
5. Advanced Architecture
   - Node Architecture
   - Situated Architecture
6. Advanced Technology
   - Middleware
   - Tools
7. Programming Tuple Centres
   - Meta-Coordination Language
   - Meta-Coordination Operations
8. Extensions
   - TuCSoN for JADE

Outline of Part III: Conclusion

9 Status of the Guide

10 Status of the Technology

11 Bibliography

Part I

Basic TuCSoN

Outline

Outline

Outline

TuCSoN

TuCSoN (Tuple Centres Spread over the Network) is a model for the coordination of distributed processes, as well as of autonomous, intelligent & mobile agents [Omicini and Zambonelli, 1999]

Main URLs

URL	`http://tucson.unibo.it/`
Bitbucket	`http://bitbucket.org/smariani/tucson/`
FaceBook	`http://www.facebook.com/TuCSoNCoordinationTechnology`

Basic Entities

- TuCSoN agents are the *coordinables*
- ReSpecT tuple centres are the (default) *coordination media* [Omicini and Denti, 2001]
- TuCSoN nodes represent the basic *topological abstraction*, which host the tuple centres
- agents, tuple centres, and nodes have *unique identities* within a TuCSoN system

System

Roughly speaking, a TuCSoN system is a collection of agents and tuple centres working together in a possibly-distributed set of nodes

Basic Interaction

- since agents are *pro-active* entities, and tuple centres are *reactive* entities, coordinables need coordination operations in order to *act* over coordination media: such operations are built out of the TuCSoN coordination language

- agents interact by exchanging tuples through tuple centres using TuCSoN coordination primitives, altogether defining the coordination language

- tuple centres provide the shared space for tuple-based communication (tuple space), along with the programmable behaviour space for tuple-based coordination (specification space)

System

Roughly speaking, a TuCSoN system is a collection of agents and tuple centres interacting in a possibly-distributed set of nodes

Basic Topology

- agents and tuple centres are spread over the network
- tuple centres belong to nodes
- agents live anywhere on the network, and can interact with the tuple centres hosted by any reachable TuCSoN node
- agents could in principle move independently of the device where they run, tuple centres are permanently associated to one device

System

Roughly speaking, a TuCSoN system is a collection of possibly-distributed nodes and agents interacting with the nodes' tuple centres

Basic Mobility

- agents could in principle *move independently* of the device where they run [Omicini and Zambonelli, 1998]
- tuple centres are essentially associated to one device, possibly *mobile*—so, tuple centre mobility is dependent on their hosting device

System

Roughly speaking, a TuCSoN system is a collection of possibly-distributed nodes, associated to possibly-mobile devices agents, interacting with the nodes' tuple centres

Outline

Nodes

- each node within a TuCSoN system is univocally identified by the pair
  $< NetworkId, PortNo >$, where
    - *NetworkId* is either the IP number or the DNS entry of the device
      hosting the node
    - *PortNo* is the port number where the TuCSoN *coordination service*
      listens to the invocations for the execution of coordination operations
- correspondingly, the abstract syntax for the identifier of a TuCSoN
 node hosted by a networked device `netid` on port `portno` is

$$netid : portno$$

Tuple Centres

- an admissible name for a tuple centre is *any* first-order ground logic term
- since each node contain at most one tuple centre for each admissible name, each tuple centre is uniquely identified by its admissible name associated to the node identifier
- the TuCSoN full name of a tuple centre tname on a node netid : portno is

$$tname @ netid : portno$$

- the full name of a tuple centre works as a tuple centre *identifier* in a TuCSoN system
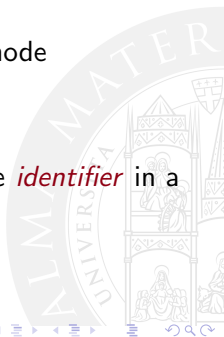
Agents

- an admissible name for an agent is *any* Prolog first-order ground logic term [Lloyd, 1984]
- when it enters a TuCSoN system, an agent assigned a *universally unique identifier* (UUID)[1]
- if an agent `aname` is assigned UUID `uuid`, its full name is

$$aname : uuid$$

---

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

Outline

Coordination Language

- the TuCSoN coordination language allows agents to interact with tuple centres by executing *coordination operations*
- TuCSoN provides coordinables with *coordination primitives*, allowing agents to read, write, consume tuples in tuple spaces, and to synchronise on them
- coordination operations are built out of coordination primitives and of the *communication languages*:
    - the tuple language
    - the tuple template language
- ! in the following, whenever unspecified, we assume that *Tuple* belongs to the tuple language, and *TupleTemplate* belongs to the tuple template language

Tuple & Tuple Template Languages

- both the tuple and the tuple template languages depend on the sort of the tuple centres adopted by TuCSoN
- given that the default TuCSoN coordination medium is the logic-based ReSpecT tuple centre, both the tuple and the tuple template languages are logic-based, too
- more precisely
  - any Prolog atom is an admissible TuCSoN tuple
  - any Prolog atom is an admissible TuCSoN tuple template
- as a result, the default TuCSoN tuple and tuple template languages coincide

Coordination Operations

- a TuCSoN *coordination operation* is invoked by a source agent on a target tuple centre, which is in charge of its execution
- any TuCSoN operation has two phases
  invocation — the request from the source agent to the target tuple centre, carrying all the information about the invocation
  completion — the response from the target tuple centre back to the source agent, including all the information about the operation execution by the tuple centre
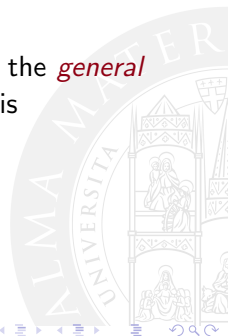
Abstract Syntax

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

$$tcid \; ? \; op$$

  where tcid is the tuple centre full name

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

$$tname \; @ \; netid : portno \; ? \; op$$

Outline

Coordination Primitives

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

- out, rd, in
- rdp, inp
- no, nop
- get, set

Basic Operations

out(*Tuple*) writes *Tuple* in the target tuple space; after the operation is successfully executed, *Tuple* is returned as a completion

rd(*TupleTemplate*) looks for a tuple matching *TupleTemplate* in the target tuple space; if a matching *Tuple* is found when the operation is first served, the execution succeeds by returning *Tuple*; otherwise, the execution is suspended, to be resumed and successfully completed when a matching *Tuple* is finally found on the target tuple space, and returned

in(*TupleTemplate*) looks for a tuple matching *TupleTemplate* in the target tuple space; if a matching *Tuple* is found when the operation is first served, the execution succeeds by removing and returning *Tuple*; otherwise, the execution is suspended, to be resumed and successfully completed when a matching *Tuple* is finally found on the target tuple space, removed, and returned

Predicative Operations

rdp(*TupleTemplate*) looks for a tuple matching *TupleTemplate* in the
target tuple space; if a matching *Tuple* is found when the
operation is served, the execution succeeds, and *Tuple* is
returned; otherwise the execution fails, and *TupleTemplate*
is returned;

inp(*TupleTemplate*) looks for a tuple matching *TupleTemplate* in the
target tuple space; if a matching *Tuple* is found when the
operation is served, the execution succeeds, *Tuple* is
removed from the target tuple space, and returned;
otherwise the execution fails, no tuple is removed from the
target tuple space, and *TupleTemplate* is returned;

Test-for-Absence Operations

no(*TupleTemplate*) looks for a *Tuple* matching *TupleTemplate* in the target tuple space; if no matching tuple is found in the target tuple space when the operation is first served, the execution succeeds, and *TupleTemplate* is returned; otherwise, the execution is suspended, to be resumed and successfully completed when no matching tuples can any longer be found in the target tuple space, then *TupleTemplate* is returned

nop(*TupleTemplate*) looks for a *Tuple* matching *TupleTemplate* in the target tuple space; if no matching tuple is found in the target tuple space when the operation is served, the execution succeeds, and *TupleTemplate* is returned; otherwise, if a matching *Tuple* is found, the execution fails, and *Tuple* is returned

Space Operations

get reads all the *Tuples* in the target tuple space, and returns them as a list; if no tuple occurs in the target tuple space at execution time, the empty list is returned, and the execution succeeds anyway

set(*Tuples*) rewrites the target tuple spaces with the list of *Tuples*; when the execution is completed, the list of *Tuples* is successfully returned
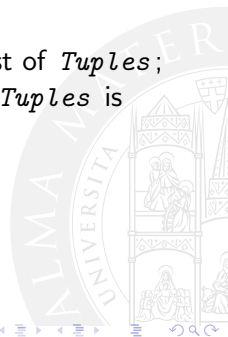
Outline

1. Basic Model & Language

2. **Basic Architecture**

3. Basic Technology

Outline

Node

- a TuCSoN system is first of all a characterised by the (possibly distributed) collection of TuCSoN nodes hosting a TuCSoN service
- a node is characterised by the networked device hosting the service, and by the network port where the TuCSoN service listens to incoming requests

Multiple nodes on a single device

Many TuCSoN nodes can in principle run on the same networked device, each one listening on a different port

Default Node

Default port

The default port number of TuCSoN is 20504

- so, an agent can invoke operations of the form

$$tname @ netid ? op$$

  without specifying the node port number `portno`, meaning that the agent intends to invoke operation `op` on the tuple centre `tname` of the default node `netid : 20504` hosted by the networked device `netid`

- any other port could in principle be used for a TuCSoN node

- the fact that a TuCSoN node is available on a networked device does *not* imply that a node is also available on the same unit on the default port—so the default node is *not* ensured to exist, generally speaking

Tuple Centres

- given an admissible tuple centre name tname, tuple centre tname is an admissibile tuple centre
- the *coordination space* of a TuCSoN node is defined as the collection of *all* the admissible tuple centres
- any TuCSoN node provides agents with a *complete* coordination space, so that in principle any coordination operation can be invoked on any admissible tuple centre belonging to any TuCSoN node
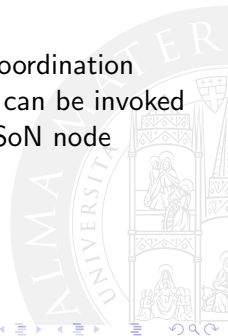
Default Tuple Centre

- every TuCSoN node defines a default tuple centre, which responds to any operation invocation received by the node that do not specify the target tuple centre

Default tuple centre

The *default tuple centre* of any TuCSoN node is named `default`

- as a result, agents can invoke operations of the form

$$@ \ netid : portno ? op$$

without specifying the tuple centre name `tname`, meaning that they intend to invoke operation `op` on the `default` tuple centre of the node `netid : portno` hosted by the networked device `netid`

Default Tuple Centre & Port

- combining the notions of default tuple centre and default port, agents can also invoke operations of the form

  @ netid ? op

  meaning that they intend to invoke operation op on the default tuple centre of the default node netid : 20504 hosted by the networked device netid
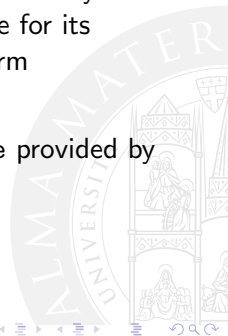
Outline

Global coordination space

- the TuCSoN global coordination space is defined at any time by the collection of all the tuple centres available on the network, hosted by a node, and identified by their full name

- a TuCSoN agent running on any networked device has at any time the whole TuCSoN global coordination space available for its coordination operations through invocations of the form

    ```
    tname @ netid : portno ? op
    ```

    which invokes operation `op` on the tuple centre `tname` provided by node `netid : portno`

Local Coordination Space

- given a networked device `netid` hosting one or more TuCSoN nodes, the TuCSoN local coordination space is defined at any time by the collection of all the tuple centres made available by all the TuCSoN nodes hosted by `netid`

- an agent running on the same device `netid` that hosts a TuCSoN node can exploit the *local coordination space* to invoke operations of the form

$$\text{tname : portno ? op}$$

  which invokes operation `op` on the tuple centre `tname` locally provided by node `netid : portno`

Defaults & Local Coordination Space

- by exploiting the notions of default node and default tuple centre, 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`

Outline

Outline

Technology Requirements

- TuCSoN is a Java-based middleware
- TuCSoN is also Prolog-based: it is based on the tuProlog Java-based technology for
  - first-order logic tuples
  - primitive & identifier parsing
  - ReSpecT specification language & virtual machine

Java & Prolog Agents

TuCSoN middleware provides

- Java API for extending Java programs with TuCSoN coordination primitives
  - package `alice.tucson.api.*`
- Java classes for programming TuCSoN agents in Java
  - `alice.tucson.api.AbstractTucsonAgent` provides a ready-to-use thread, whose main can directly use TuCSoN coordination primitives
- Prolog libraries for extending tuProlog programs with TuCSoN coordination primitives
  - `alice.tucson.api.Tucson2PLibrary` provides tuProlog agents with the ability to use TuCSoN primitives
  - by including the `:-load_library(path/to/Tucson2PLibrary)` directive in its Prolog theory

Java APIs I

Package `alice.tucson.api`

Most APIs are made available through package `alice.tucson.api`.

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

    `getAgentId():  Object` — to get the full agent ID
    `getAgentName():  String` — to get only the agent name

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

    `getAdminContext(TucsonAgentId, String, int, String, String):  AdminAC`
        — 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

Java APIs II

TucsonTupleCentreId — exposes methods to get 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 IP number
getPort(): int — to get the tuple centre host's listening 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
getLogicTupleListResult(): List<LogicTuple> — to get operation
result—to be used with bulk primitives and get_s

Java APIs III

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

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

AbstractSpawnActivity — base abstract class for user-defined TuCSoN
activities to be spawned by a spawn operation. Provides a
simplified syntax for TuCSoN operation invocations.

> doActivity(): void — to override with your spawned activity business
> logic
> out(LogicTuple): LogicTuple — out TuCSoN operation
> ... ...
> unop(LogicTuple): LogicTuple — unop TuCSoN operation

Java APIs IV

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

> `acquire_acc_1(Struct): boolean` — to get an ACC for your tuProlog
> agent
>
> `out_2(Term, Term): boolean` — out TuCSoN operation
> ... ...
> `unop_2(Term, Term): boolean` — unop TuCSoN operation

Furthermore. . .

Package `alice.tucson.api` obviously contains also all the ACCs
provided by the TuCSoN platform—among which `EnhancedACC`.
Please refer to Slides 99–105 for the complete list, and to Slide 106 for an
overview.

Java APIs V

Package `alice.logictuple`

Other APIs are made available through package `alice.logictuple`. In particular, those required to manage TuCSoN tuples.

`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

Java APIs VI

`TupleArgument` — represents TuCSoN tuples arguments (tuProlog terms), thus 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` (other similar methods provided)

    `intValue(): int` — to get the int value of the tuple argument (other similar methods provided)
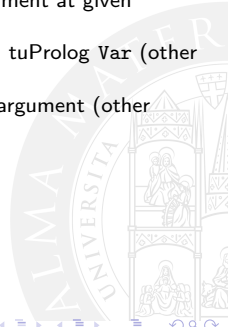
Java APIs VII

Package `alice.tucson.service`

APIs to programatically boot & kill a TuCSoN service are provided by class `TucsonNodeService` in package `alice.tucson.service`.

- constructors to init the TuCSoN service (possibly on a given port)
- methods to install & shutdown the TuCSoN service
  ```
  install():  void
  shutdown():  void
  isInstalled(String, int, int):  boolean
  ```
- entry point to launch a TuCSoN node from the command line

Java Tuples I

- TuCSoN adopts logic tuples as its main communication language
- however, Java tuples can also be used for straightforward communication among TuCSoN Java agents
- Java *tuples* and *templates* can be used
    - a Java tuple is an ordered collection of Java *tuple values*
    - a tuple value has one of the following *tuple types*: double, float, int, literal, long
    - a Java template is an ordered collection of Java *tuple values* and *tuple variables*
    - a tuple variable has either a tuple type or special type any

Java Tuples II

HelloWorld example

In `alice.tucson.examples.helloWorld.HelloWorldJTuples`, a TuCSoN agent

- builds a TuCSoN Agent Identifier, gets an ACC, and defines the TuCSoN Tuple Centre Identifier of the target tuple centre
  ```
  TucsonAgentId aid = ''helloWorldMain'';
  final NegotiationACC negAcc = TucsonMetaACC.getNegotiationContext(aid);
  final SynchACC acc = negAcc.playDefaultRole();
  final TucsonTupleCentreId tid =
      new TucsonTupleCentreId("default", "localhost", "20504");
  ```

- builds the tuple to write, and outputs it in the tuple in the tuple centre
  ```
  final IJTuple tuple = new JTuple(new JVal("hello"));
      tuple.addArg(new JVal("world"));
  ITucsonOperation op = acc.out(tid, tuple, null);
  ```

- checks the tuple in the tuple centre, by building the proper template and reading the tuple in the tuple centre, and finally releases the ACC
  ```
  final IJTupleTemplate template = new JTupleTemplate(new JVal("hello"));
  template.addArg(new JVar(JArgType.LITERAL));
  op = acc.rdp(tid, template, null); if (op.isResultSuccess()) { ... };
  ```

Java Tuples III

Main packages: tuples

IJTuple in `alice.tuples.javatuples.api` — interface representing Java tuples

JTuple in `alice.tuples.javatuples.impl` — class implementing Java tuples. Java tuples are implemented as ordered collections of `IJVal` objects

IJVal in `alice.tuples.javatuples.api` — interface representing Java tuples values (just values, not variables)

JVal in `alice.tuples.javatuples.impl` — class implementing Java tuples values. Java tuples values can be of one of the tuple types defined by enumeration `JArgType`. Methods to convert Java tuples values into primitive Java types are provided.

JArgType in `alice.tuples.javatuples.api` — enumeration defining the admissible Java tuple types: `double`, `float`, `int`, `literal`, `long`. Special type `any` is reserved for usage in Java tuple templates.

Java Tuples IV

Main packages: templates

IJTupleTemplate in `alice.tuples.javatuples.api` — interface representing Java tuple templates

JTupleTemplate in `alice.tuples.javatuples.impl` — class implementing Java tuple templates. Java tuple templates are implemented as ordered collections of `IJArg` objects.

 IJArg in `alice.tuples.javatuples.api` — interface representing both Java tuples values and Java tuple variables for templates

 IJVar in `alice.tuples.javatuples.api` — interface representing Java tuple template variables

 JVar in `alice.tuples.javatuples.impl` — class implementing Java tuple template variables. Java tuple templates variables can be of any tuple type declared in `JArgType` enumeration, including `any`. Matching of variables against templates is typed. Variables of type `any` match any value.

Java Tuples V

Main packages: operations

getJTupleResult method getJTupleResult of interface
    ITucsonOperation can be used to retrieve the result of an
    operation. It returns an object of class Tuple, which should
    then be queried if it is a tuple or a template (usual operator
    instanceof will do the job) so as to be managed
    accordingly.

Service

- given any networked device running a Java VM, a TuCSoN node can be booted to make it provide a TuCSoN service
- a TuCSoN service can be started through the `alice.tucson.service` Java API, e.g.

  ```
  java -cp tucson.jar:2p.jar alice.tucson.service.TucsonNodeService
       -portno 20506
  ```

- the node service is in charge of
  - listening to incoming operation invocations on the associated port of the device
  - dispatching them to the target tuple centres
  - returning the operation completions
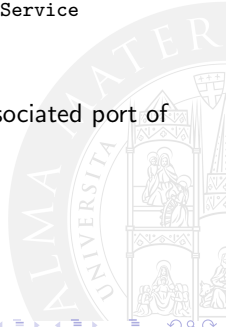
Coordination Space

- a TuCSoN node service provides the complete coordination space
- tuple centres in a node are either *actual* or *potential*: at any time in a given node

  actual tuple centres are admissible tuple centres that already *do* have a reification as a run-time abstraction

  potential tuple centres are admissible tuple centres that *do not* have a reification as a run-time abstraction, yet

- the node service is in charge of making *potential* tuple centres *actual* as soon as the first operation on them is received and served

Outline

Command Line Interpreter (CLI) I

- Shell interface for human agents / programmers, e.g.

```
java -cp tucson.jar:2p.jar
    alice.tucson.service.tools.CommandLineInterpreter
    -netid localhost -portno 20505 -aid myCLI
```

Command Line Interpreter (CLI) II

```
●●●                              lib — java — 139×48
McGriddle:lib ste$ java -cp tucson.jar:2p.jar alice.tucson.service.tools.CommandLineInterpreter -netid localhost -portno 20505 -aid myCLI
[CommandLineInterpreter]: ---------------------------------------------------------------------------------------
[CommandLineInterpreter]: Booting TuCSoN Command Line Intepreter...
[CommandLineInterpreter]: Version TuCSoN-1.11.0.0209
[CommandLineInterpreter]: ---------------------------------------------------------------------------------------
[CommandLineInterpreter]: Wed Oct 22 14:56:08 CEST 2014
[CommandLineInterpreter]: Demanding for TuCSoN default ACC on port < 20505 >...
[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)
```

TuCSoN Inspector I

A GUI tool to monitor the TuCSoN coordination space & ReSpecT VM

- to launch the Inspector tool

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

- available options are

    - `-aid` — the name of the Inspector Agent
    - `-netid` — the IP address of the device hosting the TuCSoN Node to be inspected. . .
    - `-portno` — . . . its listening port. . .
    - `-tcname` — . . . and the name of the tuplecentre to monitor

TuCSoN Inspector II

Using the Inspector Tool I

- if you launched it without specifying the full name of the target tuplecentre to inspect, choose it from the GUI
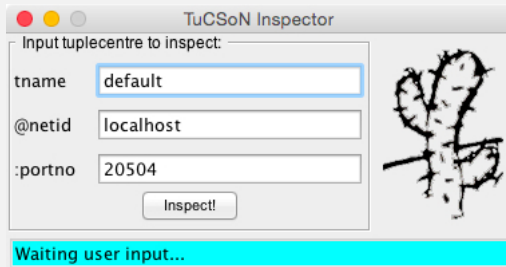
TuCSoN Inspector III

Using the Inspector Tool II

- if you launched it giving the full name of the target tuplecentre to inspect, choose what to inspect inside that tuplecentre
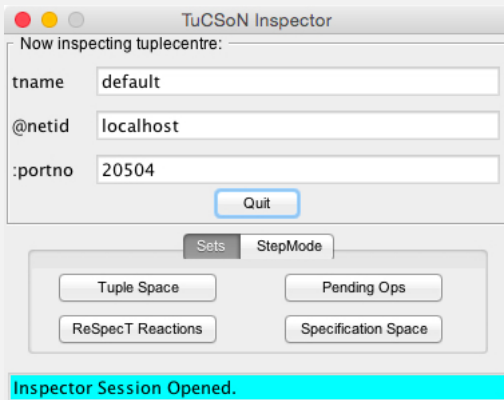
TuCSoN Inspector IV

What to inspect

In the *Sets* tab you can choose whether to inspect [a]

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 currently suspended issued operations (waiting for completion)

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

---

[a] For the *StepMode* tab, go to page 116.

TuCSoN Inspector V

Tuple Space view

In the *Tuple Space* view you can

- proactively observe the space state, thus getting any change of state, or reactively observe it, that is getting updates only when requested—through the `Observe!` button in the *Observation* tab

- filter displayed tuples according to a given admissible Prolog template—through the `Match!` button in the *Filter* tab

- dump (filtered) observations on a given log file—in the *Log* tab

TuCSoN Inspector VI
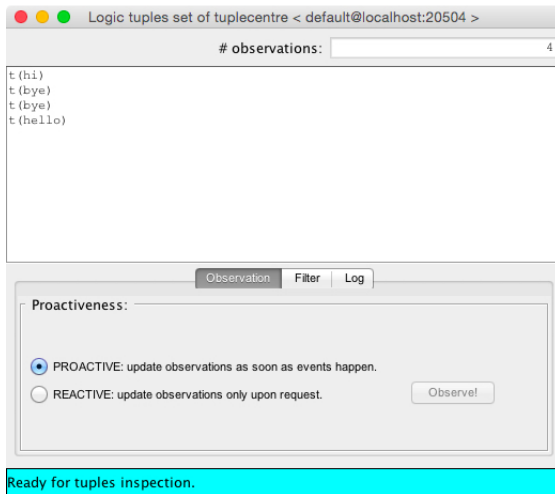
TuCSoN Inspector VII

Specification Space view

In the *Specification Space* view you can

- load a ReSpecT specification from a file...
- ...edit & set it to the current tuplecentre—through the <set_s> button
- get the ReSpecT specification from the current tuplecentre—through the <get_s> button...
- ...save it to a given file (or to the default one named default.rsp)—button Save (or Save As)

TuCSoN Inspector VIII
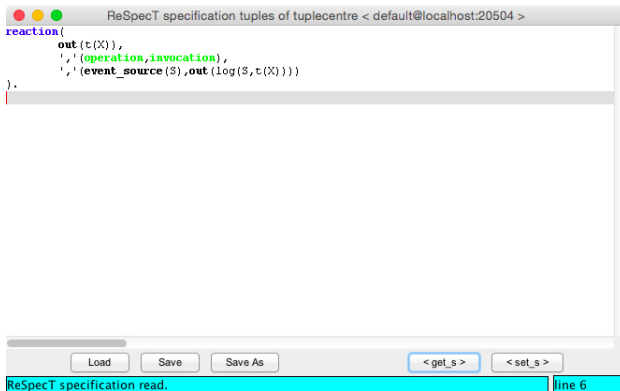
TuCSoN Inspector IX

Pending Ops view

In the *Pending Ops* view you can

- proactively observe pending TuCSoN operations, thus getting any new update whenever available, or reactively observe it, that is getting updates only when requested—through the `Observe!` button in the *Observation* tab

- filter[a] displayed TuCSoN operations according to a given admissible Prolog template—through the `Match!` button in the *Filter* tab

- dump (filtered) observations on a given log file—in the *Log* tab

---

[a]filtering is based on *operation tuples* solely a.t.m.

TuCSoN Inspector X

TuCSoN Inspector XI

ReSpecT *Reactions* view

In the ReSpecT *Reactions* view you are notified upon any ReSpecT reaction triggered in the observed tuplecentre and can dump such notifications on a given log file.


```
Triggered ReSpecT reaction set of tuplecentre < default@localhost:20504 >
reaction < reaction(out(t(ciao)),',',(event_source(_1779884064),out(log(_1779884064,t(ciao))))) > SUCCEEDED @ 15:14:27.
```

Log

store

☐   dump observations on file: inspector-reactions.log   Browse

Ready for ReSpecT reactions triggering notification.

Part II

Advanced TuCSoN

Outline

Outline

Bulk Primitives: The Idea

- bulk coordination primitives are required in order to obtain significant efficiency gains for a large class of coordination problems involving the management of more than one tuple with a single coordination operation [Rowstron, 1996]
- instead of returning one single matching tuple, bulk operations return list of matching tuples
- in case of no matching tuples, they successfully return an empty list of tuples: so, bulk primitives always succeed
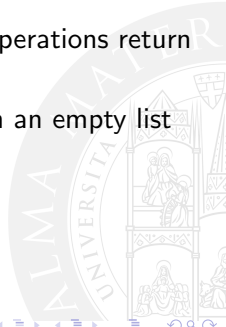
Bulk Primitives: Simple Examples

For instance, let us assume that the default tuple centre contains just 3 tuples: 2 colour(white) and 1 colour(black)

- the invocation of a rd_all(color(X)) succeeds and returns a list of 3 tuples, containing 2 colour(white) and 1 colour(black) tuples

- the invocation of a rd_all(color(black)) succeeds and returns a list of 1 tuples, containing 1 colour(black) tuples

- the invocation of a rd_all(color(blue)) succeeds and returns an empty list of tuples

- the invocation of a no_all(color(X)) succeeds and returns an empty list of tuples

- the invocation of a no_all(color(black)) succeeds and returns a list of 2 tuples, containing 2 colour(white) tuples

- the invocation of a no_all(color(blue)) succeeds and returns a list of 3 tuples, containing 2 colour(white) and 1 colour(black) tuples

On the other hand, out_all(*Tuples*) just takes a list of *Tuples* and simply put them all in the target tuple space.

Bulk Primitives in TuCSoN

The TuCSoN coordination language provides the following 4 *bulk coordination primitives* to build coordination operations

- out_all
- rd_all
- in_all
- no_all

Outline

Toward Computationally-complex Coordination

Beyond `eval`

- often, complex computational activities related to coordination – such as complex calculations, access to external structures, etc. – would be more easily expressed in terms of a "standard" sequential program executed within the coordination abstraction

- in the original LINDA, this was achieved through the `eval` primitive, which provides a sort of "expression tuple" that the tuple space evaluates based on some not-so-clear expression semantics

- the execution of the `eval` is typically reified in the tuple space in terms of a new tuple, representing the result of the (possibly complex) computational activity performed

The spawn Primitive I

Generality

- in order to allow for complex computational activities related to coordination, TuCSoN provides the spawn primitive
- spawn can activate either TuCSoN Java agent, or a tuProlog agent
- the execution of the spawn is *local* to the tuple space where it is invoked, and so are their results
    - correspondingly, the code (either Java or tuProlog) of the agent should be local to the same node hosting the tuple centre
    - also, the code can execute TuCSoN coordination primitives, but only on the same *spawning* tuple centre
- spawn semantics is *not suspensive*: it triggers a concurrent computational activity and completion is returned to the caller as soon as the concurrent activity has started

The spawn Primitive II

General syntax

- spawn has basically two parameters

  activity — a ground Prolog atom containing either the tuProlog
  theory and the goal to be solved – e.g.,
  `solve('path/to/Prolog/Theory.pl', yourGoal)` –
  or the Java class to be executed—e.g.,
  `solve('list.of.packages.YourClass.class')`

  tuple centre — a ground Prolog term identifying the target tuple
  centre that should execute the spawn

- from tuProlog, the two parameters are just the end of the story

The spawn Primitive III

Java syntax

- a third parameter is instead necessary when *spawning* from TuCSoN Java agent
- it could be either

  listener — a listener TucsonOperationCompletionListener is required for synchronous executions of spawn

  timeout — an integer value in milliseconds determining the maximum waiting time for the agent

Outline

Uniform Primitives: The Idea

- uniform coordination primitives [Gardelli et al., 2007] are required in order to inject a probabilistic mechanism within coordination, thus to obtain stochastic behaviour in coordinated systems
- 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 here performed with a *uniform distribution*
- this promote the engineering of stochastic behaviours in coordinated systems, and the implementation of nature-inspired coordination models [Omicini, 2012]

Uniform Primitives: A Simple Example

For instance, let us assume that the default tuple centre contains 15 tuples: 10 `colour(white)` and 5 `colour(black)`

- using a standard `rd(color(X))`, say, 1 billion times, don't know non-determinism ensures nothing: we could get 1 billion `colour(white)` returned, or 1 billion `colour(black)`, or any distribution in-between; the result would depend on implementation, and there is no possible *a priori* probabilistic description of the overall system behaviour

- using a uniform `urd(color(X))` in the same way, instead, ensures that at each request we have two times the chances to get `colour(white)` returned instead of `colour(black)`, and that the overall behaviour could be probabilistically described as basically returning two `colour(white)` for each `colour(black)` as the matching tuple

Uniform Primitives in TuCSoN

The TuCSoN coordination language provides the following 6 *uniform coordination primitives* to build coordination operations

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

Outline

RBAC

- Role-Based Access Control (RBAC) models integrate *organisation* and *security*
- RBAC is a NIST standard[2]
- roles are assigned to processes, and rule the distributed access to resources

---

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

RBAC in TuCSoN

- TuCSoN tuple centres are structured and ruled in organisations
- TuCSoN implements a version of RBAC [Omicini et al., 2005b], where organisation and security issues are handled in a uniform way as coordination issues
- a special tuple centre ($ORG) contains the dynamic rules of RBAC in TuCSoN
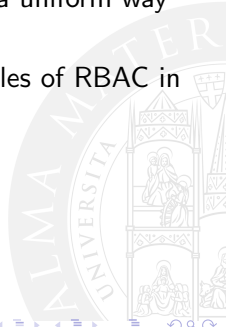
RBAC Properties in TuCSoN I

- 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 Properties in TuCSoN 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`, atm, 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 Properties in TuCSoN III

- 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`
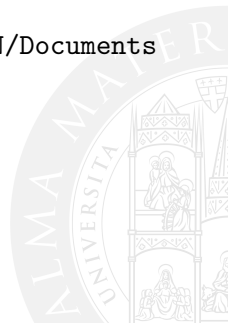
Using RBAC in TuCSoN I

- 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`:

  `getAdminContext(TucsonAgentId, String, int, String, String`
          — releases an `AdminACC`, necessary to administer
          RBAC-related properties

  `getNegotiationContext(TucsonAgentId, String, int): Negoti`
          — releases a `NegotiationACC`, necessary to activate
          roles obtaining ACCs

Using RBAC in TuCSoN II

- 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:

  `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

Using RBAC in TuCSoN III

- For other methods provided by `NegotiationACC` interface see "RBAC in TuCSoN" how-to at
  `http://apice.unibo.it/xwiki/bin/view/TuCSoN/Documents`

Outline

ACC

An Agent Coordination Context (ACC) [Omicini, 2002] is

- a runtime and stateful interface released to an agent to execute operations on the tuple centres of a specific organisation

- a sort of interface provided to an agent by the infrastructure to make it interact within a given organisation

ACC in TuCSoN

- the ACC is an organisation abstraction to model RBAC in TuCSoN [Omicini et al., 2005a]
- along with tuple centres, ACC are the run-time abstractions that allows TuCSoN to uniformly handle coordination, organisation, and security issues

Ordinary Standard ACC

OrdinarySynchACC enables standard interaction with the tuple space, and enacts a *blocking behaviour* from the agent's perspective: whichever the coordination operation invoked (either suspensive or predicative), the agent stub blocks waiting for its completion

OrdinaryAsynchACC enables standard interaction with the tuple space, and enacts a *non-blocking behaviour* from the agent's perspective: whichever the coordination operation invoked (either suspensive or predicative), the agent stub *does not block*, but is instead *asynchronously notified* of its completion

Ordinary Specification ACC

SpecificationSynchACC enables standard interaction with the
specification space and enacts a blocking behaviour from the
agent's perspective: whichever the meta-coordination
operation invoked (either suspensive or predicative), the
agent stub *blocks* waiting for its completion

SpecificationAsynchACC enables standard interaction with the
specification space and enacts a *non-blocking behaviour*
from the agent's perspective: whichever the
meta-coordination operation invoked (either suspensive or
predicative), the agent stub *does not block*, but is instead
asynchronously notified of its completion

Ordinary ACC

SynchACC enables standard interaction with both the tuple and the specification space and enacts a blocking behaviour from the agent's perspective: whichever the (meta-)coordination operation invoked (either suspensive or predicative), the agent stub *blocks* waiting for its completion

AsynchACC enables standard interaction with both the tuple and the specification space and enacts a *non-blocking behaviour* from the agent's perspective: whichever the (meta-)coordination operation invoked (either suspensive or predicative), the agent stub *does not block*, but is instead *asynchronously notified* of its completion

Bulk ACC

`BulkSynchACC` enables bulk interaction with the tuple space, and enacts a blocking behaviour from the agent's perspective: whichever the bulk coordination operation invoked, the agent stub *blocks* waiting for its completion

`BulkAsynchACC` enables bulk interaction with the tuple space, and enacts a *non-blocking behaviour* from the agent's perspective: whichever the bulk coordination operation invoked, the agent stub *does not block*, but is instead *asynchronously notified* of its completion

Uniform ACC

`UniformSynchACC` enables uniform coordination primitives with the tuple space, and enacts a blocking behaviour from the agent's perspective: whichever the uniform coordination operation invoked, the agent stub *blocks* waiting for its completion

`UniformAsynchACC` enables uniform coordination primitives with the tuple space, and enacts a *non-blocking behaviour* from the agent's perspective: whichever the uniform coordination operation invoked, the agent stub *does not block*, but is instead *asynchronously notified* of its completion

Enhanced ACC

EnhancedSynchACC enables all coordination and meta-coordination primitives, including uniform and bulk ones, with the tuple centre, and enacts a blocking behaviour from the agent's perspective: whichever the operation invoked, the agent stub *blocks* waiting for its completion

EnhancedAsynchACC enables uniform coordination primitives, including uniform and bulk ones, with the tuple centre, and enacts a *non-blocking behaviour* from the agent's perspective: whichever the bulk coordination operation invoked, the agent stub *does not block*, but is instead *asynchronously notified* of its completion

Global ACC

DefaultACC enables all coordination and meta-coordination primitives with the tuple centre, enacting both a blocking and a non-blocking behaviour from the agent's perspective

EnhancedACC enables all coordination and meta-coordination primitives, including uniform and bulk ones, with the tuple centre, enacting both a blocking and a non-blocking behaviour from the agent's perspective

Overall View over TuCSoN ACC

Outline

4. Advanced Model

5. **Advanced Architecture**

6. Advanced Technology

7. Programming Tuple Centres

8. Extensions

Outline

Architectural View of a TuCSoN Node

Outline

Situating TuCSoN I

TuCSoN coordination for environment engineering

- Distributed systems are *situated*—that is, immersed into an environment, and reactive to events of *any* sort
- Thus, coordination media are required to mediate any activity toward the environment, allowing for a fruitful interaction
- ⇒ ReSpecT tuple centres are able to *capture general environment events*, and to generally *mediate process-environment interaction*

Situating TuCSoN II

Situating TuCSoN

- Thus, *situating* TuCSoN basically means making it capable of *capturing environment events*, and *expressing general MAS-environment interactions*
  [Casadei and Omicini, 2009, Omicini and Mariani, 2013]
- ⇒ the TuCSoN middleware and the ReSpecT language
  - capture, react to, and observe general environment events
  - explicitly interact with the environment

Dealing with Environment Change I

Environment manipulation

- Source and target of a tuple centre event can be any external resource
- A suitable *identification* scheme – both at the syntax and at the infrastructure level – is introduced for environmental resources
- The coordination language is extended to express explicit manipulation of environmental resources
- New *tuple centre predicates* are introduced, whose form is
  - ⟨*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

Dealing with Environment Change II

Transducers

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

TuCSoN Situated Architecture
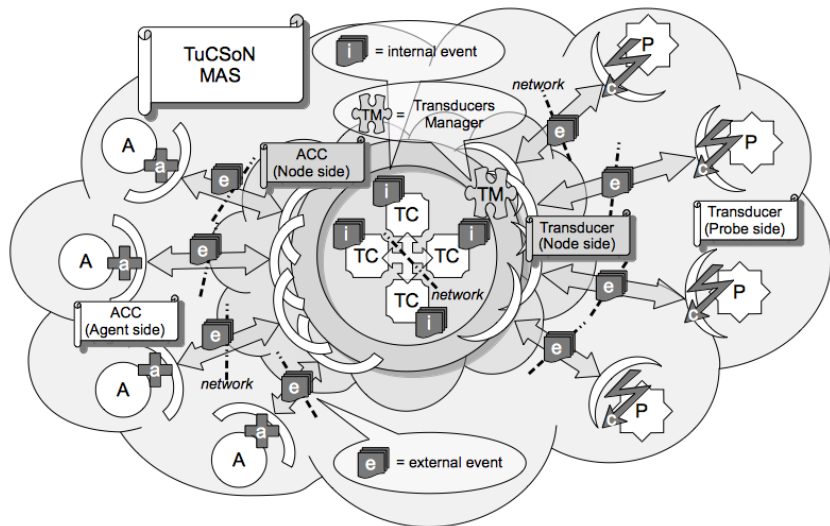
An Example: TuCSoN Thermostat

- Package `alice.tucson.examples.situatedness` contains a simple example of how to exploit TuCSoN features for situated coordination
- A step-by-step *how-to* is reported in the TuCSoN main site at

  `http://apice.unibo.it/xwiki/bin/download/TuCSoN/Documents/situatednesspdf.pdf`

Outline

Outline

Asynchronous Operation Invocation I

- Coordination operations may be invoked in two modes:

  synchronous — enforces coordination by blocking the caller agent whenever the invoked operation gets suspended

  asynchronous — enforcing coordination without hindering agent autonomy, by *decoupling* the agent control flow from the coordination operation control flow

- The asynchronous mode is now supported by a novel TuCSoN component: the `AsynchOpsHelper` in package `alice.tucson.asynchSupport`

- Agents may *delegate* invocation of coordination operations to such component, which then keeps track of *pending* and *completed* operations on their behalf

Asynchronous Operation Invocation II

- The API exposed by the `AsynchOpsHelper` consists of the following methods:

  `enqueue(AbstractTucsonAction,TucsonOperationCompletionList`
  — Adds an operation to the queue of *pending operations*

  `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* instead

- Details regarding formal and return types involved in the above
methods can be found in TuCSoN javadoc as well as in
"Asynchronous Operation Invocation in TuCSoN" how-to at
http://apice.unibo.it/xwiki/bin/view/TuCSoN/Documents

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 storing persistent information is created within the persistent/ folder *in the directory where* TuCSoN *has been installed*
- Such 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
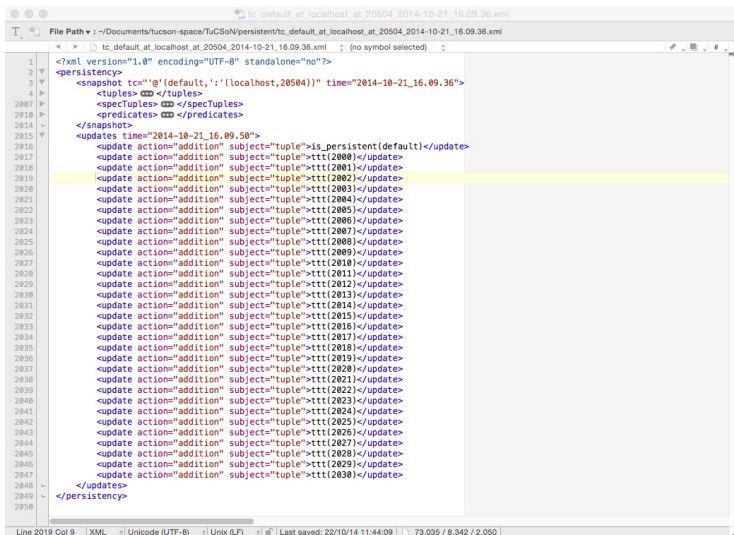
Persistency & Recovery III

- 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

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

Outline

Inspector I

Step Mode

From the *StepMode* tab of the Inspector, it is possible to (de)activate
ReSpecT VM "step mode" on the inspected tuple centre by (un)checking
the *Step Mode* checkbox:

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

Inspector II
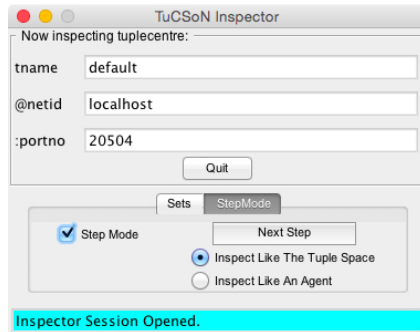
Inspector III

- 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

4 Advanced Model

5 Advanced Architecture

6 Advanced Technology

7 Programming Tuple Centres

8 Extensions

Outline

Meta-Coordination Language

- the TuCSoN meta-coordination language allows agents to program ReSpecT tuple centres by executing *meta-coordination operations*
- TuCSoN provides coordinables with *meta-coordination primitives*, allowing agents to read, write, consume ReSpecT specification tuples in tuple centres, and also to synchronise on them
- meta-coordination operations are built out of meta-coordination primitives and of the ReSpecT *specification languages*:
    - the *specification language*
    - the *specification template language*
- ! in the following, whenever unspecified, we assume that `reaction(E,G,R)` belongs to the specification language, and `reaction(ET,GT,RT)` belongs to the specification template language

Specification & Specification Template Languages

- both the specification and the specification template languages depend on the sort of the tuple centres adopted by TuCSoN
- given that the default TuCSoN coordination medium is the logic-based ReSpecT tuple centre, both the specification and the specification template languages are defined by ReSpecT
- more precisely
    - any ReSpecT reaction is an admissible TuCSoN specification tuple
    - any ReSpecT reaction is an admissible TuCSoN specification template
- as a result, the default TuCSoN specification and specification template languages coincide
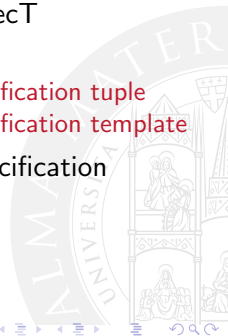
Meta-Coordination Operations

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

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

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

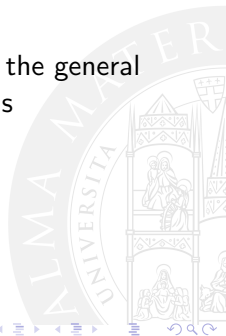# Abstract Syntax

- the abstract syntax of a coordination operation op_s invoked on a target tuple centre `tcid` is

$$\text{tcid ? } op\_\text{s}$$

  where `tcid` is the tuple centre full name

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

$$\text{tname @ netid : portno ? } op\_\text{s}$$

Outline

Meta-Coordination Primitives

- TuCSoN defines 9 meta-coordination primitives, allowing agents to read, write, consume ReSpecT specification tuples in tuple spaces, and to synchronise on them
    - rd_s, in_s, out_s
    - rdp_s, inp_s
    - no_s, nop_s
    - get_s, set_s
- meta-primitives perfectly match coordination primitives, allowing a uniform access to both the tuple space and the specification space in a TuCSoN tuple centre

Basic Meta-Operations

$out\_s(E,G,R)$ writes a specification tuple $reaction(E,G,R)$ in the target tuple centre; after the operation is successfully executed, the specification tuple is returned as a completion

$rd\_s(ET,GT,RT)$ looks for a specification tuple $reaction(E,G,R)$ matching $reaction(ET,GT,RT)$ in the target tuple centre; if a matching specification tuple is found when the operation is first served, the execution succeeds, and the matching specification tuple is returned; otherwise, the execution is suspended, to be resumed and successfully completed when a matching specification tuple is finally found on the target tuple centre, and returned

$in\_s(ET,GT,RT)$ looks for a specification tuple $reaction(E,G,R)$ matching $reaction(ET,GT,RT)$ in the target tuple centre; if a matching specification tuple is found when the operation is first served, the execution succeeds, and the matching specification tuple is removed and returned; otherwise, the execution is suspended, to be resumed and successfully completed when a matching specification tuple is finally found on the target tuple centre, removed, and returned

Predicative Meta-Operations

rdp_s(*ET*,*GT*,*RT*) looks for a specification tuple reaction(*E*,*G*,*R*)
matching reaction(*ET*,*GT*,*RT*) in the target tuple centre;
if a matching specification tuple is found when the operation
is served, the execution succeeds, and the matching
specification tuple is returned; otherwise the execution fails,
and the specification template is returned

inp_s(*ET*,*GT*,*RT*) looks for a specification tuple reaction(*E*,*G*,*R*)
matching reaction(*ET*,*GT*,*RT*) in the target tuple centre;
if a matching specification tuple is found when the operation
is served, the execution succeeds, and the matching
specification tuple is removed and returned; otherwise the
execution fails, and the specification template is returned

Test-for-Absence Meta-Operations

no_s(ET,GT,RT) looks for a specification tuple reaction(E,G,R) matching
    reaction(ET,GT,RT) in the target tuple centre—where
    reaction(ET,GT,RT) belongs to the specification template language;
    if no specification tuple is found in the target tuple centre when the
    operation is first served, the execution succeeds, and the specification
    tuple template is returned; otherwise, the execution is suspended, to be
    resumed and successfully completed when no matching specification
    tuples can any longer be found in the target tuple centre, then the
    specification tuple template is returned

nop_s(ET,GT,RT) looks for a specification tuple reaction(E,G,R) matching
    reaction(ET,GT,RT) in the target tuple centre—where
    reaction(ET,GT,RT) belongs to the specification template language;
    if no specification tuple is found in the target tuple tuple when the
    operation is first served, the execution succeeds, and the specification
    tuple template is returned; otherwise, the execution fails, and a
    matching specification tuple is returned

Space Meta-Operations

get_s reads all the specification tuples in the target tuple centre, and returns them as a list; if no specification tuple occurs in the target tuple centre at execution time, the empty list is returned, and the execution succeeds anyway

set_s([(*E1*, *G1*, *R1*), ..., (*En*, *Gn*, *Rn*)]) rewrites the target tuple spaces with the list of specification tuples reaction(*E1*, *G1*, *R1*), ..., reaction(*En*, *Gn*, *Rn*); when the execution is completed, the list of specification tuples is successfully returned

Outline

Outline

JADE

- JADE is one of the oldest and nowadays most widely used agent development frameworks [Bellifemine et al., 2007]
- JADE can be downloaded freely from `http://http://jade.tilab.com`
- Integrating TuCSoN with JADE essentially means to make coordination via tuple centres generally available to agent programmers

TuCSoN4JADE

- TuCSoN4JADE integrate TuCSoN and JADE by implementing TuCSoN as a JADE *service* [Omicini et al., 2004]
- An example of how to use TuCSoN from JADE is reported in the TuCSoN main site at `http://apice.unibo.it/xwiki/bin/download/TuCSoN/Documents/tucson4jadequickguidepdf.pdf`

Synchronous vs. Asynchronous Invocation

- The BridgeToTucson class is the component mediating all the interactions between JADE and TuCSoN

- In particular, it offers two methods for invoking coordination operations, one for each *invocation semantics* JADE agents may choose [Mariani et al., 2014]:

  synchronousInvocation() — lets agents invoke TuCSoN coordination operations *synchronously w.r.t. the caller behaviour*. This means the caller behaviour *only* is (possibly) suspended – and automatically resumed – as soon as the requested operation completes, not the agent as a whole—as in [Omicini et al., 2004].

  asynchronousInvocation() — lets clients *asynchronously* invoke TuCSoN coordination operations. Regardless of whether the coordination operation suspends, the agent does not, thus the caller behaviour continues [Mariani et al., 2014].

Part III

Conclusion

Outline

Still Missing I

Formal Semantics

- in order to fully understand and exploit TuCSoN, a full formal specification is required
- a formal specification based on [Omicini, 1999] will soon make into the TuCSoN Guide

Still Missing II

Timed & Space Coordination

- in order to fully exploit the power of tuple centres in the engineering of complex computational systems, the ReSpecT language should be fully described, both syntactical and semantically
- its main extensions toward
  - timed coordination [Omicini et al., 2005c]
  - spatial coordination [Mariani and Omicini, 2013]

should be described in the TuCSoN Guide along the same line as situated coordination
[Casadei and Omicini, 2009, Omicini and Mariani, 2013]

Still Missing III

Semantic Coordination

- in order to exploit TuCSoN within knowledge-intensive environments, semantic tuple centres were defined [Nardini et al., 2012]
- the resulting *Semantic* TuCSoN *coordination model* should be described in the TuCSoN Guide

Outline

Still Missing I

Space-time Coordination & Situatedness

- the current implementation of timed extension of ReSpecT tuple centres is stable and reliable, however its documentation is delegated to the forthcoming ReSpecT documentation
- the current implementation of spatial extension of ReSpecT tuple centres is stable, not yet released, but available upon request
- the current implementation of situatedness is now completed

Semantic Coordination

- a working implementation of Semantic TuCSoN is available, but not yet integrated with the current TuCSoN implementation

Outline

9 Status of the Guide

10 Status of the Technology

11 Bibliography

Bibliography I

Bellifemine, F. L., Caire, G., and Greenwood, D. (2007).
Developing Multi-Agent Systems with JADE.
Wiley.

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

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

Lloyd, J. W. (1984).
Foundations of Logic Programming.
Springer, 1st edition.

Bibliography II

Mariani, S. and Omicini, A. (2013).
Space-aware coordination in ReSpecT.
In Baldoni, M., Baroglio, C., Bergenti, F., and Garro, A., editors, *From Objects to Agents*, volume 1099 of *CEUR Workshop Proceedings*, pages 1–7, Turin, Italy. Sun SITE Central Europe, RWTH Aachen University.
XIV Workshop (WOA 2013). Workshop Notes.

Mariani, S., Omicini, A., and Sangiorgi, L. (2014).
Models of autonomy and coordination: Integrating subjective & objective approaches in agent development frameworks.
In Braubach, L., Camacho, D., and Venticinque, S., editors, *8th International Symposium on Intelligent Distributed Computing (IDC 2014)*, Madrid, Spain.

Nardini, E., Omicini, A., and Viroli, M. (2012).
Semantic tuple centres.
Science of Computer Programming.
Special Issue on Self-Organizing Coordination.

Bibliography III

📄 Omicini, A. (1999).
On the semantics of tuple-based coordination models.
In *1999 ACM Symposium on Applied Computing (SAC'99)*, pages 175–182, New York, NY, USA. ACM.
Special Track on Coordination Models, Languages and Applications.

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

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

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

Bibliography IV

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

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

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

Bibliography V

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

Omicini, A., Ricci, A., Viroli, M., Cioffi, M., and Rimassa, G. (2004).
Multi-agent infrastructures for objective and subjective coordination.
Applied Artificial Intelligence: An International Journal, 18(9–10):815–831.
Special Issue: Best papers from EUMAS 2003: The 1st European Workshop on Multi-agent Systems.

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

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

Bibliography VI

Rowstron, A. I. T. (1996).
Bulk Primitives in Linda Run-Time Systems.
PhD thesis, The University of York.

The TuCSoN Coordination Model & Technology
A Guide

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

Alma Mater Studiorum—Università di Bologna a Cesena

TuCSoN v. 1.12.0.0301
Guide v. 1.3.1
June 15, 2015