Outline of Part I: Basic TuCSoN

1. Basic Model & Language
   - Basic Model
   - Naming
   - Basic Language
   - Basic Operations

2. Basic Architecture
   - Nodes & Tuple Centres
   - Coordination Spaces

3. Basic Technology
   - Middleware
   - Tools
Outline of Part II: Advanced TuCSoN

4 Advanced Model
- Bulk Primitives
- Coordinative Computation
- Uniform Primitives
- Organisation
- Agent Coordination Contexts

5 Advanced Architecture
- Node Architecture

6 Programming Tuple Centres
- Meta-Coordination Language
- Meta-Coordination Operations
Outline of Part III: Conclusion

7 Status of the Guide

8 Status of the Technology

9 Bibliography
Part I

Basic TuCSoN
Outline

1. Basic Model & Language
2. Basic Architecture
3. Basic Technology
Part 1: Basic TuCSoN

1. Basic Model & Language
   - Basic Model
     - Naming
     - Basic Language
     - Basic Operations

2. Basic Architecture
   - Nodes & Tuple Centres
   - Coordination Spaces

3. Basic Technology
   - Middleware
   - Tools
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

<table>
<thead>
<tr>
<th>URL</th>
<th><a href="http://tucson.apice.unibo.it/">http://tucson.apice.unibo.it/</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>FaceBook</td>
<td><a href="http://www.facebook.com/TuCSoNCoordinationTechnology">http://www.facebook.com/TuCSoNCoordinationTechnology</a></td>
</tr>
<tr>
<td>Google Code</td>
<td><a href="http://tucson.googlecode.com/">http://tucson.googlecode.com/</a></td>
</tr>
<tr>
<td>SourceForge</td>
<td><a href="http://sf.net/projects/tucson/">http://sf.net/projects/tucson/</a></td>
</tr>
</tbody>
</table>
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
Part 1: Basic TuCSoN

1. Basic Model & Language
   - Basic Model
   - Naming
   - Basic Language
   - Basic Operations

2. Basic Architecture
   - Nodes & Tuple Centres
   - Coordination Spaces

3. Basic Technology
   - Middleware
   - Tools
Nodes

- each node within a TuCSoN system is univocally identified by the pair \(<\text{NetworkId}, \text{PortNo}\>)\), where
  - \text{NetworkId} is either the IP number or the DNS entry of the device hosting the node
  - \text{PortNo} is the port number where the TuCSoN \textit{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 \textit{netid} on port \textit{portno} is
  \[
  \text{netid} : \text{portno}
  \]
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
  
  \[ tname \otimes \text{netid : portno} \]

- 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)
- if an agent `aname` is assigned UUID `uuid`, its full name is `aname : uuid`

1http://docs.oracle.com/javase/7/docs/api/java/util/UUID.html
Part 1: Basic TuCSoN

1. Basic Model & Language
   - Basic Model
   - Naming
   - Basic Language
   - Basic Operations

2. Basic Architecture
   - Nodes & Tuple Centres
   - Coordination Spaces

3. Basic Technology
   - Middleware
   - Tools
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 \( \text{op} \) invoked on a target tuple centre \( \text{tcid} \) is

\[
\text{tcid} \ ? \ \text{op}
\]

where \( \text{tcid} \) 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

\[
\text{tname} \ @ \ \text{netid} : \ \text{portno} \ ? \ \text{op}
\]

Omicini, Mariani (Università di Bologna)
Part 1: Basic TuCSoN

1. Basic Model & Language
   - Basic Model
   - Naming
   - Basic Language
   - Basic Operations

2. Basic Architecture
   - Nodes & Tuple Centres
   - Coordination Spaces

3. Basic Technology
   - Middleware
   - Tools
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(\textit{Tuples})** rewrites the target tuple spaces with the list of *Tuples*; when the execution is completed, the list of *Tuples* is successfully returned.
Part 1: Basic TuCSoN

1. Basic Model & Language
   - Basic Model
   - Naming
   - Basic Language
   - Basic Operations

2. Basic Architecture
   - Nodes & Tuple Centres
   - Coordination Spaces

3. Basic Technology
   - Middleware
   - Tools
Node

- a TuCSoN system is first of all 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.
The default port number of TuCSoN is 20504.

- so, an agent can invoke operations of the form
  
  \[ \text{tname @ netid ? op} \]

  without specifying the node port number \text{portno}, meaning that the
  agent intends to invoke operation \text{op} on the tuple centre \text{tname} of the
  default node \text{netid} : 20504 hosted by the networked device \text{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
  \textit{not} imply that a node is also available on the same unit on the default
  port—so the default node is \textit{not} ensured to exist, generally speaking
Tuple Centres

- given an admissible tuple centre name \( tname \), tuple centre \( tname \) is an admissible 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`.
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
Part 1: Basic TuCSoN

1. Basic Model & Language
   - Basic Model
   - Naming
   - Basic Language
   - Basic Operations

2. Basic Architecture
   - Nodes & Tuple Centres
   - Coordination Spaces

3. Basic Technology
   - Middleware
   - Tools
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:
  \[\text{tname @ netid} : \text{portno} \, ? \, \text{op}\]
  which invokes operation \text{op} on the tuple centre \text{tname} provided by node \text{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

  \[
  \text{tname} : \text{portno} \ ? \text{op}
  \]

  which invokes operation `op` on the tuple centre `tname` locally provided by node `netid : portno`
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 \texttt{op} on the default tuple centre of node netid : portno
- tname ? op
  invoking operation \texttt{op} on the tname tuple centre of default node netid : 20504
- op
  invoking operation \texttt{op} on the default tuple centre of default node netid : 20504
Part 1: Basic TuCSoN

1 Basic Model & Language
   - Basic Model
   - Naming
   - Basic Language
   - Basic Operations

2 Basic Architecture
   - Nodes & Tuple Centres
   - Coordination Spaces

3 Basic Technology
   - Middleware
   - Tools
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.TucsonAgent 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 an ACC. The ACC is mandatory to interact with a TuCSoN tuple centre.

- `getContext(TucsonAgentId, String, int)`: EnhancedACC — to get an 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

**TucsonAgent** — 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

**SpawnActivity** — 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.

- `get_context_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 85–91 for the complete list, and to Slide 92 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)` — to encode a given string into a TuCSoN tuple/template
- `getName()` — to get the functor name of the tuple
- `getArg(int)` — to get the tuple argument at given position
**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)
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
  
  ```java
  install(): void
  shutdown(): void
  ```
- entry point to launch a TuCSoN node from the command line

2 Always an EnhancedACC in current implementation TuCSoN-1.10.3.0206
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-1.10.3.0206.jar alice.tucson.service.TucsonNodeService -port 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.
Part 1: Basic TuCSoN

1. Basic Model & Language
   - Basic Model
   - Naming
   - Basic Language
   - Basic Operations

2. Basic Architecture
   - Nodes & Tuple Centres
   - Coordination Spaces

3. Basic Technology
   - Middleware
   - Tools
Shell interface for human agents / programmers, e.g.

```
java -cp TuCSoN-1.10.3.0206.jar
alice.tucson.service.tools.CommandLineInterpreter
-netid localhost -port 20505 -aid myCLI
```
# Command Line Interface (CLI) II

## CLI Syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\langle TucsonOp \rangle ::= \langle TcName \rangle @ \langle IpAddress \rangle : \langle PortNo \rangle ? \langle Op \rangle</td>
<td></td>
</tr>
<tr>
<td>\langle TcName \rangle ::= Prolog ground term</td>
<td></td>
</tr>
<tr>
<td>\langle IpAddress \rangle ::= localhost</td>
<td>IP address</td>
</tr>
<tr>
<td>\langle PortNo \rangle ::= port number</td>
<td></td>
</tr>
<tr>
<td>\langle Op \rangle ::= out(T)</td>
<td>in(TT)</td>
</tr>
<tr>
<td>T,T1,...,Tn ::= tuple (Prolog term)</td>
<td></td>
</tr>
<tr>
<td>TT ::= tuple template (Prolog term)</td>
<td></td>
</tr>
<tr>
<td>TL ::= list of tuples (Prolog list of terms)</td>
<td></td>
</tr>
<tr>
<td>E,E1,...,En ::= ReSpecT event</td>
<td></td>
</tr>
<tr>
<td>G,G1,...,Gn ::= ReSpecT guard predicate</td>
<td></td>
</tr>
<tr>
<td>R,R1,...,Rn ::= ReSpecT reaction body</td>
<td></td>
</tr>
<tr>
<td>ET ::= ReSpecT event template</td>
<td></td>
</tr>
<tr>
<td>GT ::= ReSpecT guard template</td>
<td></td>
</tr>
<tr>
<td>RT ::= ReSpecT reaction body template</td>
<td></td>
</tr>
</tbody>
</table>
A GUI tool to monitor the TuCSoN coordination space & ReSpecT VM—to some extent, actually it’s still in development

- to launch the Inspector tool
  
  java -cp TuCSoN-1.10.3.0206.jar alice.tucson.introspection.tools.Inspector

- 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.
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.

![Inspector Tool II](image)
TuCSoN Inspector IV

What to inspect

In the Sets tab you can choose whether to inspect:

- **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
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

Logic tuples set of tuplecentre < default@localhost:20504 >

# observations: 8

- temperature:room:1,1, celsius:15
- temperature:room:1,2, celsius:17
- temperature:room:2,1, celsius:16
- temperature:room:2,2, celsius:14
- humidity:room:1,1, percentage:30
- humidity:room:1,2, percentage:27
- humidity:room:2,1, percentage:22
- humidity:room:2,2, percentage:35

Output:

- dump observations on file: inspector-tuples.log

Template:

- Filter observed tuples using the following template:

Ready for tuples inspection.
**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

```
reaction
  out: precious\{X\},
    \',\' completion, success\},
  out: backup\{X\})
}.

reaction
  in\{X\},
    \',\' completion, success\},
  out: consumed\{X\})
}.
```
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

Pending TuCSoN operations set of tuplecentre < default@localhost:20504 >

# observations: 2

in::operation::req::1), who::cli01), res::R)) from <cli01> to '<@::default,::localhost,20504)'>
in::operation::req::1), who::cli02), res::R)) from <cli02> to '<@::default,::localhost,20504)'>

Filtering

- Filter observed tuples using the following template:

Ready for pending operations inspection.
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.
TuCSOn Inspector XII

Triggered ReSpecT reaction set of tuplecentre < default@localhost:20504 >

reaction < reaction.out.precious.pin.1234)), out.backup.pin.1234))) > SUCCEEDED @ 9:20:44.
reaction < reaction.in.precious.pin.1234), out.consumed.precious.pin.1234))) > SUCCEEDED @ 9:20:52.

Store

dump observations on file: inspector-reactions.log

Ready for ReSpecT reactions triggering notification.
Part II

Advanced TuCSoN
Part 2: Advanced TuCSoN

4 Advanced Model
- Bulk Primitives
  - Coordinative Computation
  - Uniform Primitives
  - Organisation
  - Agent Coordination Contexts

5 Advanced Architecture
- Node Architecture

6 Programming Tuple Centres
- Meta-Coordination Language
- Meta-Coordination Operations
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 \texttt{colour(white)} and 1 \texttt{colour(black)}

- the invocation of a \texttt{rd.all(color(X))} succeeds and returns a list of 3 tuples, containing 2 \texttt{colour(white)} and 1 \texttt{colour(black)} tuples

- the invocation of a \texttt{rd.all(color(black))} succeeds and returns a list of 1 tuples, containing 1 \texttt{colour(black)} tuples

- the invocation of a \texttt{rd.all(color(blue))} succeeds and returns an empty list of tuples

- the invocation of a \texttt{no.all(color(X))} succeeds and returns an empty list of tuples

- the invocation of a \texttt{no.all(color(black))} succeeds and returns a list of 2 tuples, containing 2 \texttt{colour(white)} tuples

- the invocation of a \texttt{no.all(color(blue))} succeeds and returns a list of 3 tuples, containing 2 \texttt{colour(white)} and 1 \texttt{colour(black)} tuples

On the other hand, \texttt{out.all(Tuples)} just takes a list of \texttt{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
Part 2: Advanced TuCSoN

4 Advanced Model
- Bulk Primitives
- Coordinative Computation
- Uniform Primitives
- Organisation
- Agent Coordination Contexts

5 Advanced Architecture
- Node Architecture

6 Programming Tuple Centres
- Meta-Coordination Language
- Meta-Coordination Operations
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 $\text{spawn}$ Primitive I

**Generality**

- In order to allow for complex computational activities related to coordination, TuCSoN provides the $\text{spawn}$ primitive.
- $\text{spawn}$ can activate either TuCSoN Java agent, or a tuProlog agent.
- The execution of the $\text{spawn}$ 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.
- $\text{spawn}$ 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
Part 2: Advanced TuCSoN

4 Advanced Model
- Bulk Primitives
- Coordinative Computation
- Uniform Primitives
- Organisation
- Agent Coordination Contexts

5 Advanced Architecture
- Node Architecture

6 Programming Tuple Centres
- Meta-Coordination Language
- Meta-Coordination Operations
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 \texttt{colour(white)} and 5 \texttt{colour(black)}

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

- using a uniform \texttt{urd(color(X))} in the same way, instead, ensures that at each request we have two times the chances to get \texttt{colour(white)} returned instead of \texttt{colour(black)}, and that the overall behaviour could be probabilistically described as basically returning two \texttt{colour(white)} for each \texttt{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
Part 2: Advanced TuCSoN

Advanced Model
- Bulk Primitives
- Coordinative Computation
- Uniform Primitives
- Organisation
  - Agent Coordination Contexts

Advanced Architecture
- Node Architecture

Programming Tuple Centres
- Meta-Coordination Language
- Meta-Coordination Operations
RBAC

- Role-Based Access Control (RBAC) models integrate organisation and security
- RBAC is a NIST standard\(^3\)
- roles are assigned to processes, and rule the distributed access to resources

\(^3\)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.

Current TuCSoN implementation does not provide a stable and reliable implementation of RBAC, yet...
Part 2: Advanced TuCSoN

4 Advanced Model
- Bulk Primitives
- Coordinative Computation
- Uniform Primitives
- Organisation
- Agent Coordination Contexts

5 Advanced Architecture
- Node Architecture

6 Programming Tuple Centres
- Meta-Coordination Language
- Meta-Coordination Operations
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
- current TuCSoN implementation provide a limited yet useful implementation of the ACC notion
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
Part 2: Advanced TuCSoN

4 Advanced Model
- Bulk Primitives
- Coordinative Computation
- Uniform Primitives
- Organisation
- Agent Coordination Contexts

5 Advanced Architecture
- Node Architecture

6 Programming Tuple Centres
- Meta-Coordination Language
- Meta-Coordination Operations
Architectural View of a TuCSoN Node
Part 2: Advanced TuCSoN

4 Advanced Model
- Bulk Primitives
- Coordinative Computation
- Uniform Primitives
- Organisation
- Agent Coordination Contexts

5 Advanced Architecture
- Node Architecture

6 Programming Tuple Centres
- Meta-Coordination Language
- Meta-Coordination Operations
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.
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
  \[ tcid ? \, op_s \]
  where $tcid$ is the tuple centre full name
- given the structure of the full name of a tuple centre, the general abstract syntax of a TuCSoN coordination operation is
  \[ \text{tname} \, @ \, \text{netid} : \, \text{portno} \, ? \, \, op_s \]
Part 2: Advanced TuCSoN

4 Advanced Model
- Bulk Primitives
- Coordinative Computation
- Uniform Primitives
- Organisation
- Agent Coordination Contexts

5 Advanced Architecture
- Node Architecture

6 Programming Tuple Centres
- Meta-Coordination Language
- Meta-Coordination Operations
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

\[ \text{rdp}_s(ET, GT, RT) \] looks for a specification tuple \( \text{reaction}(E, G, R) \) matching \( \text{reaction}(ET, GT, RT) \) 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.

\[ \text{inp}_s(ET, GT, RT) \] looks for a specification tuple \( \text{reaction}(E, G, R) \) matching \( \text{reaction}(ET, GT, RT) \) 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

\[ \text{no}_s(ET, GT, RT) \] looks for a specification tuple \( \text{reaction}(E, G, R) \) matching \( \text{reaction}(ET, GT, RT) \) in the target tuple centre—where \( \text{reaction}(ET, GT, RT) \) 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

\[ \text{nop}_s(ET, GT, RT) \] looks for a specification tuple \( \text{reaction}(E, G, R) \) matching \( \text{reaction}(ET, GT, RT) \) in the target tuple centre—where \( \text{reaction}(ET, GT, RT) \) 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.
Part III

Conclusion
### 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

#### Organisation & Security
- in order to fully exploit integration of organisation and security with coordination, a complete specification of Agent Coordination Contexts and RBAC in TuCSoN is required
- model, architecture, and specification of ACC and RBAC are required to complete the TuCSoN Guide
Still Missing II

Timed Coordination & Situatedness
- 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]
  - situatedness [Casadei and Omicini, 2009]
should be described in the TuCSoN Guide

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
Still Missing I

Organisation & Security

- the TuCSoN technology does not provide a stable and reliable implementation of RBAC, yet
- the current implementation of ACC provides a limited yet useful implementation of the ACC notion

Timed 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
- situatedness still not fully implemented neither in ReSpecT (language extensions) nor in the TuCSoN infrastructure (transducers)
Still Missing II

Semantic Coordination

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


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. 1.10.3.0206
Guide v. 1.0.2
January 9, 2013