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

Outline of Part I: Basic TuCSoN

Outline of Part II: Advanced TuCSoN

Outline of Part III: Conclusion

Part I

Basic TuCSoN

Outline

Part 1: Basic TuCSoN

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.apice.unibo.it/
FaceBook	http://www.facebook.com/TuCSoNCoordinationTechnology
Google Code	http://tucson.googlecode.com/
SourceForge	http://sf.net/projects/tucson/

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

Nodes

- each node within a TuCSoN system is univocally identified by the pair $< \mathit{NetworkId}, \mathit{PortNo} >$, 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

Part 1: Basic TuCSoN

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`

Part 1: Basic TuCSoN

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

Part 1: Basic TuCSoN

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`

Part 1: Basic TuCSoN

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

  `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`

Part 1: Basic TuCSoN

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.[2] 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): 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
  ```

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

[2] Always an `EnhancedACC` in current implementationTuCSoN-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

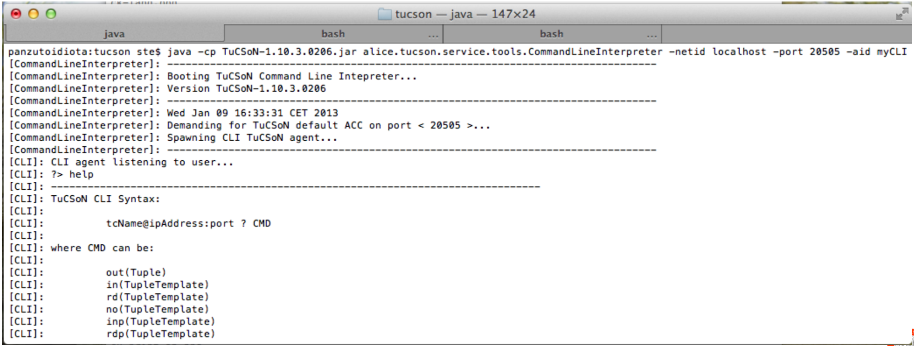
  potential tuple centres — are admissible tuple centres that *do not* have a reification as a run-time abstraction, yet
- the node service is in charge of making *potential* tuple centres *actual* as soon as the first operation on them is received and served

Part 1: Basic TuCSoN

Command Line Interface (CLI) I

- 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

⟨*TucsonOp*⟩	::=	⟨*TcName*⟩ `@` ⟨*IpAddress*⟩ `:` ⟨*PortNo*⟩ `?` ⟨*Op*⟩
⟨*TcName*⟩	::=	Prolog ground term
⟨*IpAddress*⟩	::=	`localhost` | IP address
⟨*PortNo*⟩	::=	port number
⟨*Op*⟩	::=	`out(T)` | `in(TT)` | `rd(TT)` | `no(TT)` |`inp(TT)` | `rdp(TT)` | `nop(TT)` | `get()` | `set([T1,...,Tn])` | `out_all(TT,TL)` | `in_all(TT,TL)` | `rd_all(TT,TL)` | `no_all(TT,TL)` | `uin(TT)` | `urd(TT)` | `uno(TT)` | `uinp(TT)` | `urdp(TT)` | `unop(TT)` | `out_s(E,G,R)` | `in_s(ET,GT,RT)` | `rd_s(ET,GT,RT)` | `no_s(ET,GT,RT)` | `inp_s(ET,GT,RT)` | `rdp_s(ET,GT,RT)` | `nop_s(ET,GT,RT)` | `get_s()` | `set_s([(E1,G1,R1),...,(En,Gn,Rn)])`
T,T1,...,Tn	::=	tuple (Prolog term)
TT	::=	tuple template (Prolog term)
TL	::=	list of tuples (Prolog list of terms)
E,E1,...,En	::=	ReSpecT event
G,G1,...,Gn	::=	ReSpecT guard predicate
R,R1,...,Rn	::=	ReSpecT reaction body
ET	::=	ReSpecT event template
GT	::=	ReSpecT guard template
RT	::=	ReSpecT reaction body template

TuCSoN Inspector I

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...

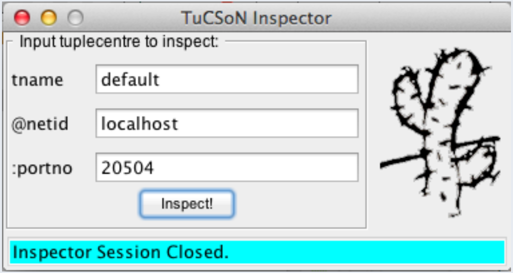
`-portno` — ...its listening port...

`-tcname` — ...and the name of the tuplecentre to monitor

TuCSoN Inspector II

Using the Inspector Tool I

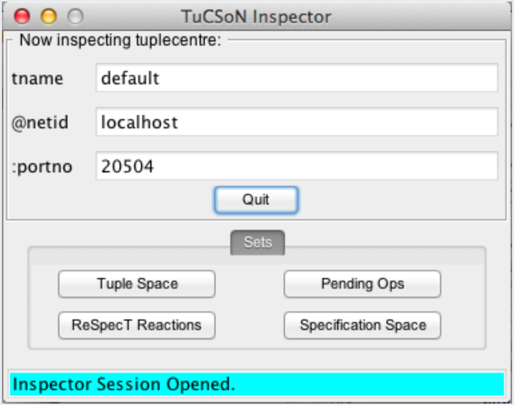
- if you launched it without specifying the full name of the target tuplecentre to inspect, choose it from the GUI



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

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

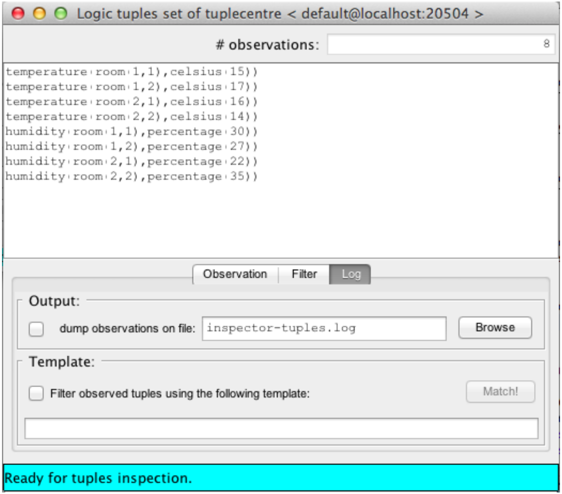
TuCSoN Inspector V

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



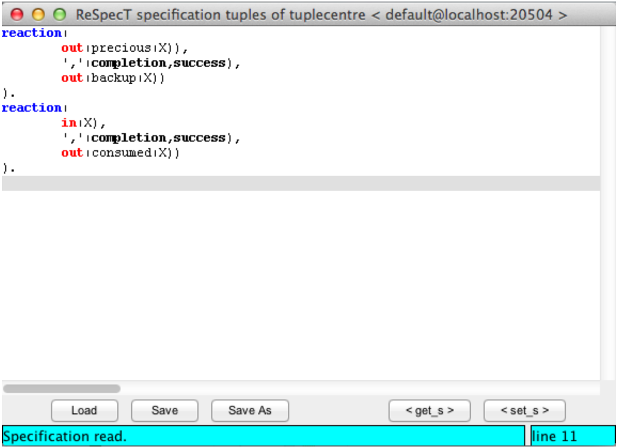
TuCSoN Inspector VII

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

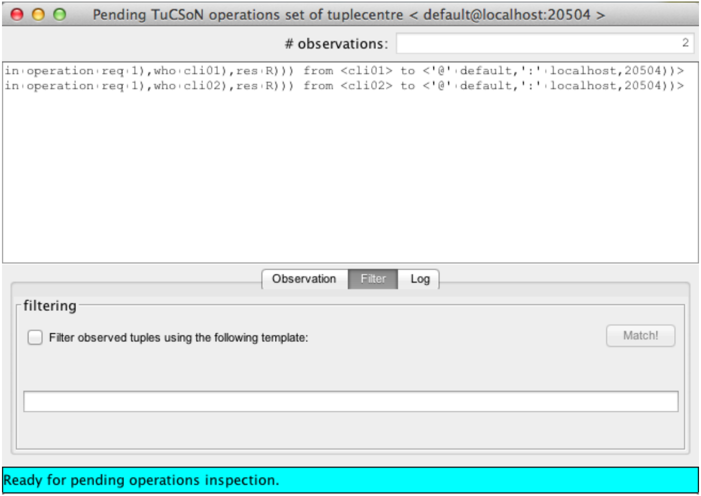
Pending Ops view

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

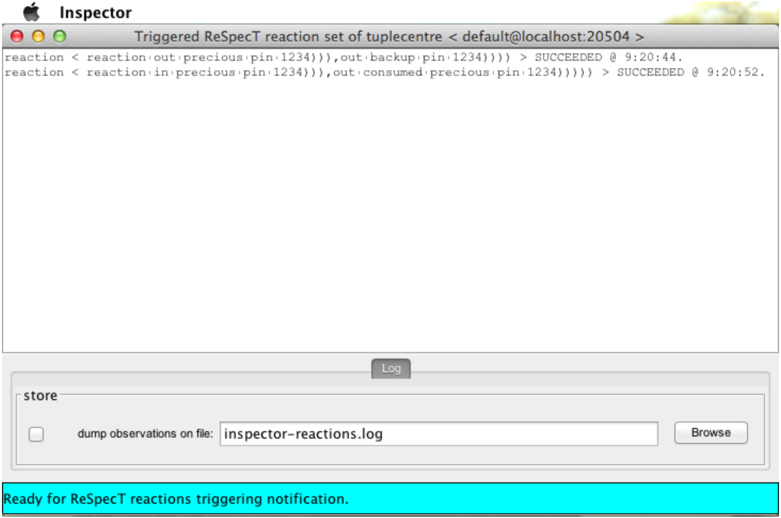


TuCSoN Inspector XI

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.

TuCSoN Inspector XII



Part II

Advanced TuCSoN

Part 2: Advanced TuCSoN

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

Part 2: Advanced TuCSoN

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

Part 2: Advanced TuCSoN

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`

Part 2: Advanced TuCSoN

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

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

! 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

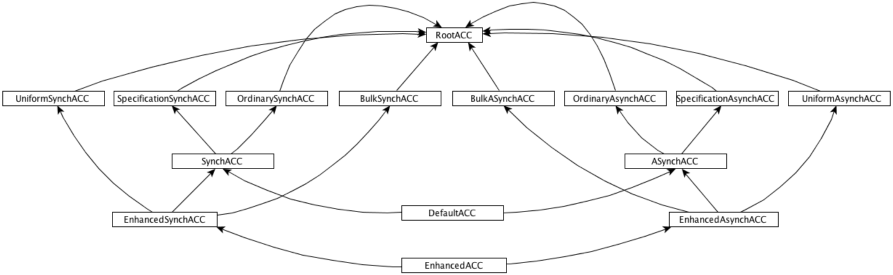
`EnhancedAsynchACC` enables uniform coordination primitives, including uniform and bulk ones, with the tuple centre, and enacts a *non-blocking behaviour* from the agent's perspective: whichever the bulk coordination operation invoked, the agent stub *does not block*, but is instead *asynchronously notified*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

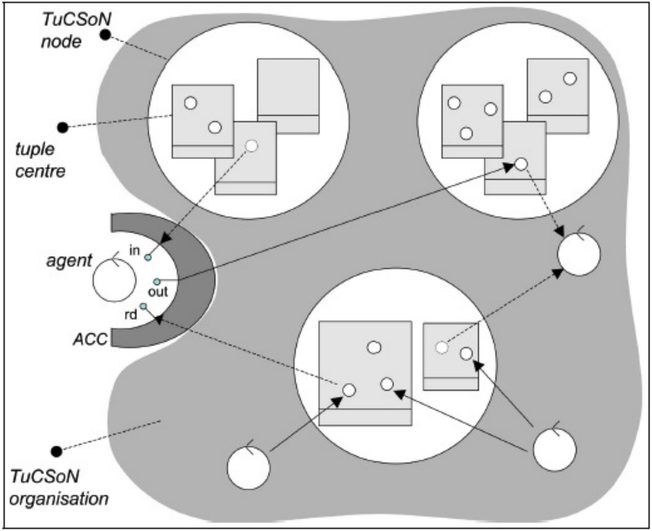
`EnhancedACC` enables all coordination and meta-coordination primitives, including uniform and bulk ones, with the tuple centre, enacting both a blocking and a non-blocking behaviour from the agent's perspective

Overall View over TuCSoN ACC



Part 2: Advanced TuCSoN

Architectural View of a TuCSoN Node



Part 2: Advanced TuCSoN

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

  `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

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

Part 2: Advanced TuCSoN

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

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

Bibliography I

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

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.

Bibliography II

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

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

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.

Bibliography III

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., 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. 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. 1.10.3.0206
Guide v. 1.0.2
January 9, 2013