TuCSoN: Tuple Centres Spread over the Network ADVANCED

Marco Fabbri
(after Andrea Omicini, Stefano Mariani)
marco.fabbri28@unibo.it

Dipartimento di Informatica: Scienza e Ingegneria (DISI)
ALMA MATER STUDIORUM—Università di Bologna

DS 2013/2014
Distributed Systems – 22nd of May 2014
DISCLAIMER

These slides are an excerpt from the official TuCSoN guide available at http://www.slideshare.net/andreaomicini/the-tucson-coordination-model-technology-a-guide by Andrea Omicini and Stefano Mariani.

Then, credits for all the stuff (text & images) goes to the original authors.

Credits for all the mistakes goes to the author.
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
- Situated 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 colour(white) and 1 colour(black)

- the invocation of a \texttt{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 \texttt{rd\_all(color(black))} succeeds and returns a list of 1 tuples, containing 1 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 colour(white) tuples
- the invocation of a \texttt{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, \texttt{out\_all(Tuples)} just takes a list of \textit{Tuples} and simply put them all in the target tuple space.
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
- Situated Architecture

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

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

5 Advanced Architecture
- Node Architecture
- Situated Architecture

6 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
- roles are assigned to processes, and rule the distributed access to resources

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 ($\textit{ORG}$) 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
   - Situated 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

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

Advanced Architecture
- Node Architecture
- Situated Architecture

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
- Situated Architecture

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

Omicini, Mariani (Università di Bologna)
TuCSoN coordination for environment engineering

- Distributed systems are *situated*—that is, immersed into an environment, and reactive to events of *any* sort
- Thus, coordination media are required to mediate any activity toward the environment, allowing for a fruitful interaction

⇒ ReSpecT tuple centres are able to *capture general environment events*, and to generally *mediate process-environment interaction*
Thus, **situating** TuCSoN basically means making it capable of *capturing environment events*, and *expressing general MAS-environment interactions* [Casadei and Omicini, 2009, Omicini and Mariani, 2013]

⇒ the TuCSoN middleware and the ReSpecT language
- capture, react to, and observe general environment events
- explicitly interact with the environment
Dealing with Environment Change I

Environment manipulation

- Source and target of a tuple centre event can be any external resource.
- A suitable *identification* scheme – both at the syntax and at the infrastructure level – is introduced for environmental resources.
- The coordination language is extended to express explicit manipulation of environmental resources.
- New *tuple centre predicates* are introduced, whose form is:
  - \( \langle EResId \rangle ? \text{get}(\langle Key \rangle,\langle Value \rangle) \)
    enabling a tuple centre to get properties of environmental resources.
  - \( \langle EResId \rangle ? \text{set}(\langle Key \rangle,\langle Value \rangle) \)
    enabling a tuple centre to set properties of environmental resources.
Dealing with Environment Change II

Transducers

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

Diagram showing an architecture with agents, tuple centres, transducers, and resources.
An Example: TuCSoN Thermostat

- Package `alice.tucson.examples.situatedness` contains a simple example of how to exploit TuCSoN features for situated coordination.
- A step-by-step *how-to* is reported in the TuCSoN main site at http://apice.unibo.it/xwiki/bin/view/TuCSoN/DocumentsSituatednessHowTo
Part 2: Advanced TuCSoN

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

5 Advanced Architecture
  - Node Architecture
  - Situated Architecture

6 Programming Tuple Centres
  - Meta-Coordination Language
  - Meta-Coordination Operations
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
  
  $$tname \ @ \ netid : \ portno \ ? \ op_s$$
Part 2: Advanced TuCSoN

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

5 Advanced Architecture
- Node Architecture
- Situated Architecture

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

\[ \text{out}_s(E,G,R) \] writes a specification tuple \( \text{reaction}(E,G,R) \) in the target tuple centre; after the operation is successfully executed, the specification tuple is returned as a completion.

\[ \text{rd}_s(ET,GT,RT) \] 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 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.

\[ \text{in}_s(ET,GT,RT) \] 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 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

\texttt{no_s(ET,GT,RT)} looks for a specification tuple \texttt{reaction(E,G,R)} matching \texttt{reaction(ET,GT,RT)} in the target tuple centre—where \texttt{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

\texttt{nop_s(ET,GT,RT)} looks for a specification tuple \texttt{reaction(E,G,R)} matching \texttt{reaction(ET,GT,RT)} in the target tuple centre—where \texttt{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.


An algebraic approach for modelling organisation, roles and contexts in MAS.
Special Issue: Process Algebras and Multi-Agent Systems.

RBAC for organisation and security in an agent coordination infrastructure.

Time-aware coordination in ReSpecT.

Coordination of mobile information agents in TuCSoN.
*Internet Research, 8*(5):400–413.

TuCSoN: Tuple Centres Spread over the Network ADVANCED

Marco Fabbri
(after Andrea Omicini, Stefano Mariani)
marco.fabbri28@unibo.it

Dipartimento di Informatica: Scienza e Ingegneria (DISI)
ALMA MATER STUDIORUM—Università di Bologna

DS 2013/2014
Distributed Systems – 22nd of May 2014