Experiments in TuCSoN
Distributed Systems
Sistemi Distribuiti

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

Ingegneria Due
Alma Mater Studiorum—Università di Bologna a Cesena

Academic Year 2011/2012
Outline of Part I: Basic TuCSoN

1. Basic Model & Language
2. Basic Architecture
3. Basic Technology
4. Basic Experiments
   - Examples
Outline of Part II: Advanced TuCSoN

5. Advanced Model
6. Advanced Architecture
7. Programming Tuple Centres
8. Experiments in ReSpecT
Outline of Part III: Conclusion

9 Conclusion & Perspectives

10 Bibliography
Part I

Basic TuCSoN
Outline

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

1. Basic Model & Language

2. Basic Architecture

3. Basic Technology

4. Basic Experiments
   - Examples
TuCSoN Coordination Model I

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]

URL http://tucson.apice.unibo.it/
Basic Entities

- TuCSoN agents are the *coordinables*
- ReSpecT tuple centres are the *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
- 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).

- 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
- roughly speaking, a TuCSoN system is a collection of possibly-distributed nodes and agents interacting with the nodes’ tuple centres
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 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 $\text{netid}$ on port $\text{portno}$ is
  $$\text{netid} : \text{portno}$$
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* first-order ground logic term
- 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`

*a*http://docs.oracle.com/javase/7/docs/api/java/util/UUID.html
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*

- coordination operations are invoked by agents upon tuple centres, which are then to be univocally referred in the operation
a TuCSoN coordination operation is invoked by a source agent on a target tuple centre, which is in charge of its execution.

the abstract syntax of a coordination operation op invoked on a target tuple centre whose full name is tcid is

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

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

\[ \text{tname} \ @ \ \text{netid} : \ \text{portno} \ ? \ \text{op} \]
The TuCSoN coordination language provides 8 *coordination primitives* to build coordination operations:

- out, rd, in
- rdp, inp
- no
- get, set
The TuCSoN Basic Model & Language

TuCSoN Coordination Operations I

Basic Operations

out(Tuple) writes Tuple in the target tuple space—where Tuple belongs to the tuple language.

rd(TupleTemplate) reads a Tuple matching TupleTemplate in the target tuple space—where TupleTemplate belongs to the tuple template language; if such a tuple is not found when the operation is first served, the execution is suspended, to be resumed and completed when a matching Tuple is finally found on the target tuple space, and returned.

in(TupleTemplate) consumes a Tuple matching TupleTemplate from the target tuple space—where TupleTemplate belongs to the tuple template language; if such a tuple is not found when the operation is first served, the execution is suspended, to be resumed and completed when a matching Tuple is finally found on the target tuple space, and returned.
Predicative Operations

\texttt{rdp(TupleTemplate)} reads a \textit{Tuple} matching \textit{TupleTemplate} in the target tuple space—where \textit{TupleTemplate} belongs to the tuple template language; if such a tuple is not found when the operation is served, the execution fails, and the operation results in a failure; otherwise the operation succeeds, and \textit{Tuple} is returned.

\texttt{inp(TupleTemplate)} consumes a \textit{Tuple} matching \textit{TupleTemplate} from the target tuple space—where \textit{TupleTemplate} belongs to the tuple template language; if such a tuple is not found when the operation is served, the execution fails, and the operation results in a failure; otherwise the operation succeeds, and \textit{Tuple} is returned.
Test-for-Absence Operation

\texttt{no(TupleTemplate)} reads a \textit{Tuple} matching \textit{TupleTemplate} in the target tuple space—where \textit{TupleTemplate} belongs to the tuple template language; if a matching \textit{Tuple} is found when the operation is served, the execution fails, and \textit{Tuple} is returned; otherwise the operation succeeds.

Space Operations

\texttt{get()} reads all the tuples in the target tuple space, and returns them as a list.
\texttt{set(Tuples)} rewrites the target tuple spaces with the list of \textit{Tuples}. 
Part 1: Basic TuCSoN

1. Basic Model & Language

2. Basic Architecture

3. Basic Technology

4. Basic Experiments
   - Examples
TuCSoN Nodes & Tuple Centres I

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
- 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
  \[ tname @ netid ? op \]
  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.
TuCSoN Nodes & Tuple Centres III

Tuple Centres

- given an admissible tuple centre name \texttt{tname}, tuple centre \texttt{tname} is an admissible tuple centre
- the \textit{coordination space} of a TuCSoN node is defined as the collection of \textit{all} the admissible tuple centres
- any TuCSoN node provides agents with a \textit{complete} coordination space, so that in principle any coordination operation can be invoked on any admissible tuple centre belonging to any TuCSoN node
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

! 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

\[ @ \text{netid} ? \text{op} \]

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.
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

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

  which invokes operation op on the tuple centre tname provided by node netid : portno
The TuCSoN Basic Architecture

TuCSoN Coordination Spaces II

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 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

1. Basic Model & Language
2. Basic Architecture
3. Basic Technology
4. Basic Experiments
   - Examples
TuCSoN Middleware I

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
  - Prolog libraries for extending Prolog programs with TuCSoN coordination primitives—in particular, tuProlog programs
  - Java classes for programming TuCSoN agents in Java
TuCSoN Service

- given any networked device running a Java VM, a TuCSoN node service can be booted through the alice.tucson.service Java API
  - e.g. `java -cp TuCSoN-1.9.10.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
TuCSoN Middleware III

TuCSoN 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.
The TuCSoN Basic Technology

TuCSoN Tools I

Command Line Interface (CLI)

- shell interface for human agents / programmers
- e.g.

```java
java -cp TuCSoN-1.9.10.jar alice.tucson.service.tools.CommandLineInterpreter
   -netid localhost -port 20506 -aid myCLI
```

Inspector

- a GUI tool to monitor the TuCSoN coordination space
- e.g.

```java
java -cp TuCSoN-1.9.10.jar alice.tucson.introspection.tools.Inspector
```
TuCSoN Tools II

TuCSoN Inspector

tuple centre information

name: default
node: localhost
port: 20504

inspect
TuCSoN Tools III
TuCSoN Tools IV

```
Tuple Set of default@localhost:20505

vm time 13381993861
local time 13381993862
items 2

tuple_centre{default}:
t{hi}
```

**Observation**

- get any new observation
- get only when update requested

**Ready**
TuCSoN Tools VII

[Image of a software interface with buttons labeled 'Load', 'Save', 'Save As', 'Test', 'Get', 'Set']

Specification read.
Part 1: Basic TuCSoN

1. Basic Model & Language
2. Basic Architecture
3. Basic Technology
4. Basic Experiments
   - Examples
Where to Get the Examples

- all the files used in the next slides can be found at
  http://apice.unibo.it/xwiki/bin/view/Courses/Sd1112Lab-Class5
Example 1: CLI Operations I

**CLI Experiments**

- get bash file `launch.sh`

- launch a node, e.g. `bash launch.sh Node`, or
  
  `java -cp TuCSoN-1.9.10.jar alice.tucson.service.TucsonNodeService`

- launch the CLI tool, e.g. `bash launch.sh CLI`, or
  
  `java -cp TuCSoN-1.9.10.jar alice.tucson.service.tools.CommandLineInterpreter`

- then, experiment with the TuCSoN primitives via CLI, which provides a TuCSoN interface to human agents

- its syntax is then the standard TuCSoN syntax for coordination primitives
Example 1: CLI Operations II

**CLI Syntax**

- `out(T) --> success [ / failure]`
- `in(TT), rd(TT) --> success: Tuple [ / failure]`
- `inp(TT), rdp(TT) --> success: Tuple / failure`
- `no(TT) --> success / failure: Tuple`
- `get() --> tuple space: [Tuple1, ..., TupleN]`
- `set([T1, ..., TN]) --> success [ / failure]`
- `out_s(E,G,R) --> success [ / failure]`
- `in_s(ET,GT,RT), rd_s(ET,GT,RT) --> success: reaction(E,G,R) [ / failure]`
- `inp_s(ET,GT,RT), rdp_s(ET,GT,RT) --> success: reaction(E,G,R) / failure`
- `no_s(ET,GT,RT) --> success / failure: reaction(E,G,R)`
- `get_s() --> specification space: [reaction(E1,G1,R1), ..., reaction(En,Gn,Rn)]`
- `set_s([(E1,G1,B1), ..., (En,Gn,Bn)]) --> success [ / failure]`
Example 1: CLI Operations III

CLI Example

1. `out(msg("Hello World!"))`
2. `rd(msg(Message))`
3. `get()`
4. `in(msg(Message))`
5. `get()`
Example 2: Hello World from Java

```java
public class HelloWorld {
    public static void main(String[] args) {
        TucsonAgentId me = null;
        TucsonTupleCentrId ttcid = null;
        try {
            me = new TucsonAgentId("helloAgent");
            ttcid = new TucsonTupleCentrId("default", "localhost", "20504");
        } catch (TucsonInvalidAgentIdException e) {
            e.printStackTrace();
        } catch (TucsonInvalidTupleCentrIdException e) {
            e.printStackTrace();
        }
        SynchOnlyACC acc = TucsonMetaACC.getContext(me, "localhost", 20504);
        long now = System.currentTimeMillis();
        LogicTuple tuple = new LogicTuple("msg", new Value("Hello World!"),
            new Value("time", new Value(now)));
        try {
            acc.out(ttcid, tuple, null);
            System.out.println("Tuple inserted: " + tuple);
            LogicTuple template = new LogicTuple("msg", new Var("Msg"),
                new Var("Time"));
            LogicTuple msg = acc.in(ttcid, template, null);
            System.out.println("Tuple retrieved name: " + msg.getName());
            System.out.println("Msg argument: " + msg.getArg(0));
            System.out.println("Time argument: " + msg.getArg(1).getArg(0));
            acc.exit();
        } catch (TucsonOperationNotPossibleException e) {
            e.printStackTrace();
        } catch (UnreachableNodeException e) {
            e.printStackTrace();
        } catch (OperationTimeOutException e) {
            e.printStackTrace();
        } catch (InvalidTupleOperationException e) {
            e.printStackTrace();
        }
    }
}
```
Example 3: Hello World from Java with TucsonAgent

```java
public class HelloAgent extends TucsonAgent{
    protected HelloAgent(String aid) throws TucsonInvalidAgentIdException {
        super(aid);
    }

    @Override
    protected void main() {
        TucsonTupleCentrId ttcid = null;
        try {
            ttcid = new TucsonTupleCentrId("default", "localhost", "20504");
        } catch (TucsonInvalidTupleCentrIdException e) {
            e.printStackTrace();
        }

        SynchronousACM acc = getController();
        long now = System.currentTimeMillis();
        try {
            LogicTuple tuple = LogicTuple.parse("msg("Hello World!", time(" + now + "))\n\n            acc.out(ttcid, tuple, null);
            say("Tuple inserted: " + tuple);
            LogicTuple template = LogicTuple.parse("msg(Msg, Time)\n\n            LogicTuple msg = acc.in(ttcid, template, null);
            say("Tuple retrieved name: " + msg.getName());
            say("Msg argument: " + msg.getArg(0));
            say("Time argument: " + msg.getArg(1).getArg(0));
            acc.exit();
        } catch (TucsonOperationNotPossibleException e) {
            e.printStackTrace();
        } catch (UnreachableNodeException e) {
            e.printStackTrace();
        } catch (OperationTimeoutException e) {
            e.printStackTrace();
        } catch (InvalidTupleException e) {
            e.printStackTrace();
        } catch (InvalidLogicTupleException e) {
            e.printStackTrace();
        }
    }
}
```
Example 3: Hello World from Java with TucsonAgent II

```java
package tucson.examples.hello_agent;

import alice.tucson.api.exceptions.TucsonInvalidAgentIdException;

public class HelloAgentTest {

    /**
     * @param args
     */
    public static void main(String[] args) {
        try {
            new HelloAgent("helloAgent").spawn();
        } catch (TucsonInvalidAgentIdException e) {
            e.printStackTrace();
        }
    }
}
```
Running Examples 2 & 3

- check whether your TuCSoN node is still alive
- get examples.zip
- open Eclipse, and create a new Java project
- there, import unzipped example.zip
- run tucson.examples.hello.HelloWorld
- check your TuCSoN node
- run tucson.examples.hello_agent.HelloAgentTest
- check your TuCSoN node
Part II

Advanced TuCSoN
Part 2: Advanced TuCSoN

5 Advanced Model

6 Advanced Architecture

7 Programming Tuple Centres

8 Experiments in ReSpecT
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/
TuCSoN Organisation II

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 ($\text{ORG}$) contains the dynamic rules of RBAC in TuCSoN

! the current TuCSoN implementation provides an unstable and unreliable implementation of RBAC
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
- the current TuCSoN implementation provide a limited yet useful implementation of the ACC notion
Currently Available ACC

**OrdinarySynchACC** enables 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.

**SpecificationSynchACC** enables 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.

**OrdinaryAsynchACC** enables 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.

**SpecificationAsynchACC** enables 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.
Part 2: Advanced TuCSoN

5 Advanced Model

6 Advanced Architecture

7 Programming Tuple Centres

8 Experiments in ReSpecT
TuCSoN System
Part 2: Advanced TuCSoN

5 Advanced Model

6 Advanced Architecture

7 Programming Tuple Centres

8 Experiments in ReSpecT
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*

Meta-coordination operations are invoked by agents upon tuple centres, which are then to be univocally referred in the operation.
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.
- The abstract syntax of a coordination operation $op\_s$ invoked on a target tuple centre whose full name is $tcid$ is:
  \[
  tcid ? op\_s
  \]
- Given the structure of the full name of a tuple centre, the general abstract syntax of a TuCSoN coordination operation is:
  \[
  tname @ netid : portno ? op\_s
  \]
TuCSoN defines 8 meta-coordination primitives, allowing agents to read, write, consume ReSpecT specification tuples in tuple spaces, and to synchronise on them:

- out_s
- rd_s, in_s
- rdp_s, inp_s
- no_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.
TuCSoN Meta-Coordination Operations I

Basic Meta-Operations

\texttt{out}_s(E,G,R) \text{ writes a specification tuple } reaction(E,G,R) \text{ in the target tuple centre—where } reaction(E,G,R) \text{ belongs to the specification language}

\texttt{rd}_s(ET,GT,RT) \text{ reads a specification tuple } reaction(E,G,R) \text{ matching } reaction(ET,GT,RT) \text{ in the target tuple centre—where } reaction(ET,GT,RT) \text{ belongs to the specification template language; if such a specification tuple is not found when the operation is first served, the execution is suspended, to be resumed and completed when a matching } reaction(E,G,R) \text{ specification tuple is finally found on the target tuple centre, and returned}

\texttt{in}_s(ET,GT,RT) \text{ consumes a specification tuple } reaction(E,G,R) \text{ matching } reaction(ET,GT,RT) \text{ in the target tuple centre—where } reaction(ET,GT,RT) \text{ belongs to the specification template language; if such a specification tuple is not found when the operation is first served, the execution is suspended, to be resumed and completed when a matching } reaction(E,G,R) \text{ specification tuple is finally found on the target tuple centre, and returned}
TuCSoN Meta-Coordination Operations II

Predicative Meta-Operations

`rdp_s(ET,GT,RT)` reads 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 such a specification tuple is not found when the operation is served, the execution fails, and the operation results in a failure; otherwise the operation succeeds, and `reaction(E,G,R)` is returned.

`inp_s(ET,GT,RT)` consumes 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 such a specification tuple is not found when the operation is served, the execution fails, and the operation results in a failure; otherwise the operation succeeds, and `reaction(E,G,R)` is returned.
**Test-for-Absence Meta-Operation**

\[ \text{no}_s(ET, GT, RT) \] reads 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 a matching \( \text{reaction}(E, G, R) \) specification tuple is found when the operation is served, the execution fails, and \( \text{reaction}(E, G, R) \) is returned; otherwise the operation succeeds.

**Space Meta-Operations**

\[ \text{get}_s() \] reads all the specification tuples in the target tuple centre, and returns them as a list.

\[ \text{set}_s([[E_1, G_1, R_1], \ldots, (E_n, G_n, R_n)]) \] rewrites the target tuple spaces with the list of specification tuples \( \text{reaction}(E_1, G_1, R_1), \ldots, \text{reaction}(E_n, G_n, R_n) \).
Part 2: Advanced TuCSoN

5. Advanced Model

6. Advanced Architecture

7. Programming Tuple Centres

8. Experiments in ReSpecT
Experiments in ReSpecT Tuple Centre Programming

Programming Tuple Centres from TuCSoN CLI

CLI Example

1. `out(msg("Hello World!"))`
2. `rd(msg(Message))`
3. `out_s(in(msg(Message)),completion,out(notice("Message",Message,"removed"))))`
4. `get_s()`
5. `in(msg(Message))`
6. `get()`
Programming Tuple Centres from TuCSoN Agents

Example of a TuCSoN System

- check whether your TuCSoN node is still alive
- go back to Eclipse
- run tucson.examples.programmability.BagOfTaskTest
- check your TuCSoN node
Part III

Conclusion
Basic Coordination Middleware

- A Java-based coordination middleware for distributed process coordination
- basic tools for monitoring the coordination space

Advanced Coordination Middleware

- integrating organisation and security with coordination
- tuple centre programming for advanced coordination
Beyond TuCSoN

- ReSpecT: an assembly language for interaction / coordination
- TuCSoN: an advanced platform for experiments in
  - knowledge-based coordination
  - semantic coordination
  - adaptive & self-organising coordination
Part 3: Conclusion

9 Conclusion & Perspectives

10 Bibliography


Experiments in TuCSoN
Distributed Systems
Sistemi Distribuiti

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

Ingegneria Due
ALMA MATER STUDIORUM—Università di Bologna a Cesena

Academic Year 2011/2012