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

Outline of Part II: Advanced TuCSoN

Outline of Part III: Conclusion

Part I

Basic TuCSoN

Outline

Part 1: Basic TuCSoN

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/`

TuCSoN Coordination Model II

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

TuCSoN Coordination Model III

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

TuCSoN Coordination Model IV

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

TuCSoN Naming I

Nodes

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

```
netid : portno
```

TuCSoN Naming II

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

TuCSoN Naming III

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)[a]
- 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`

TuCSoN Coordination Language I

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

TuCSoN Coordination Language II

Coordination Operations

- 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

```
tcid ? op
```

- 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
```

TuCSoN Coordination Language III

Coordination Primitives

The TuCSoN coordination language provides 8 *coordination primitives* to build coordination operations:

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

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

TuCSoN Coordination Operations II

Predicative Operations

`rdp(`*`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 served, the execution fails, and the operation results in a failure; otherwise the operation succeeds, and *Tuple* is returned

`inp(`*`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 served, the execution fails, and the operation results in a failure; otherwise the operation succeeds, and *Tuple* is returned

TuCSoN Coordination Operations III

Test-for-Absence Operation

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

Space Operations

`get()` reads all the tuples in the target tuple space, and returns them as a list

`set(`*`Tuples`*`)` rewrites the target tuple spaces with the list of *Tuples*

Part 1: Basic TuCSoN

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

TuCSoN Nodes & Tuple Centres II

Default Node

- ! 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 `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

TuCSoN Nodes & Tuple Centres IV

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

! 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`

TuCSoN Nodes & Tuple Centres V

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`

TuCSoN Coordination Spaces I

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`

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

  `tname : portno ? op`

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

TuCSoN Coordination Spaces III

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

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 Middleware II

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

TuCSoN Tools I

Command Line Interface (CLI)

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

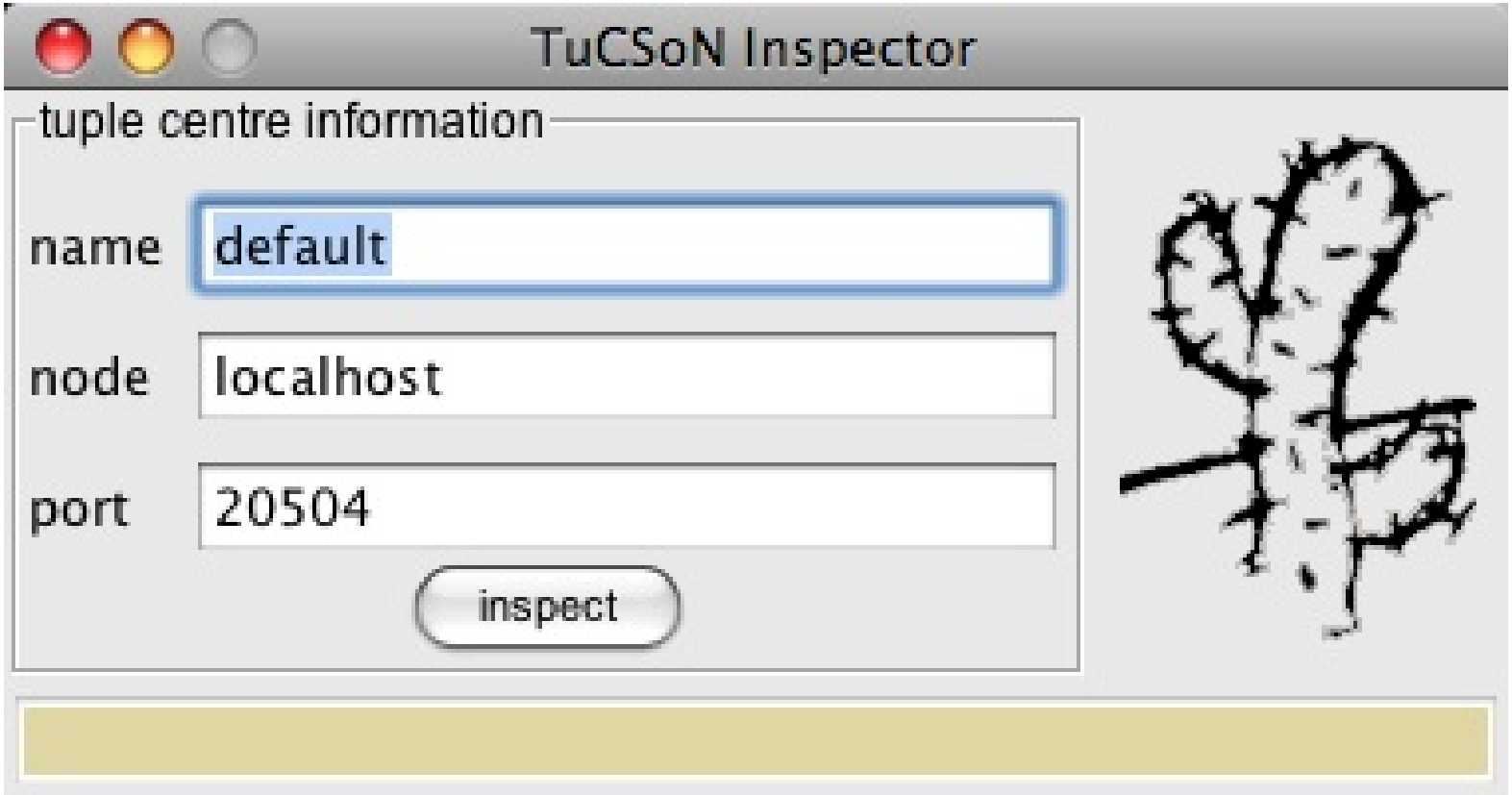
```
java -cp TuCSoN-1.9.10.jar
alice.tucson.service.tools.CommandLineInterpreter
    -netid localhost -port 20506 -aid myCLI
```

Inspector

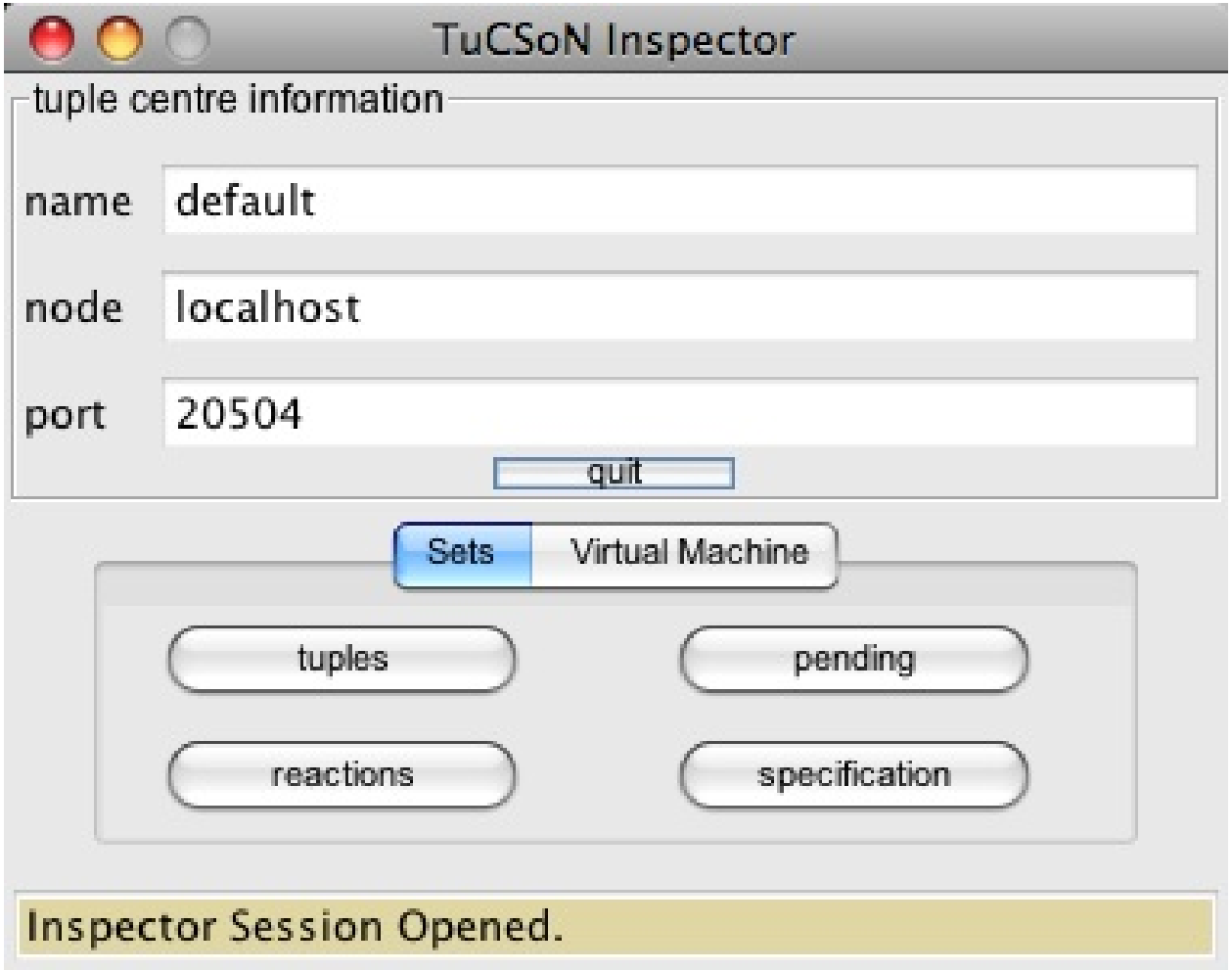
- a GUI tool to monitor the TuCSoN coordination space
- e.g.

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

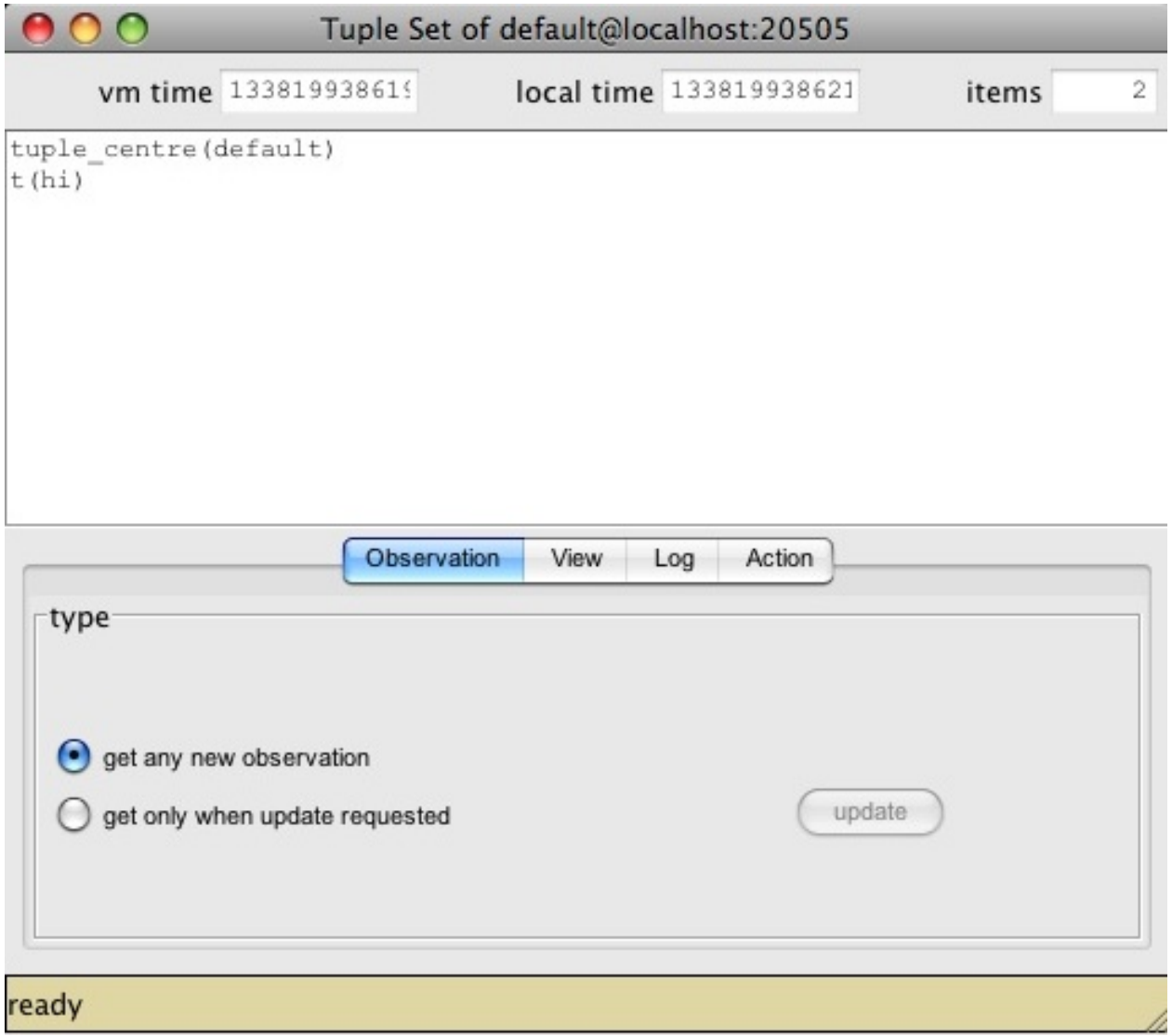
TuCSoN Tools II



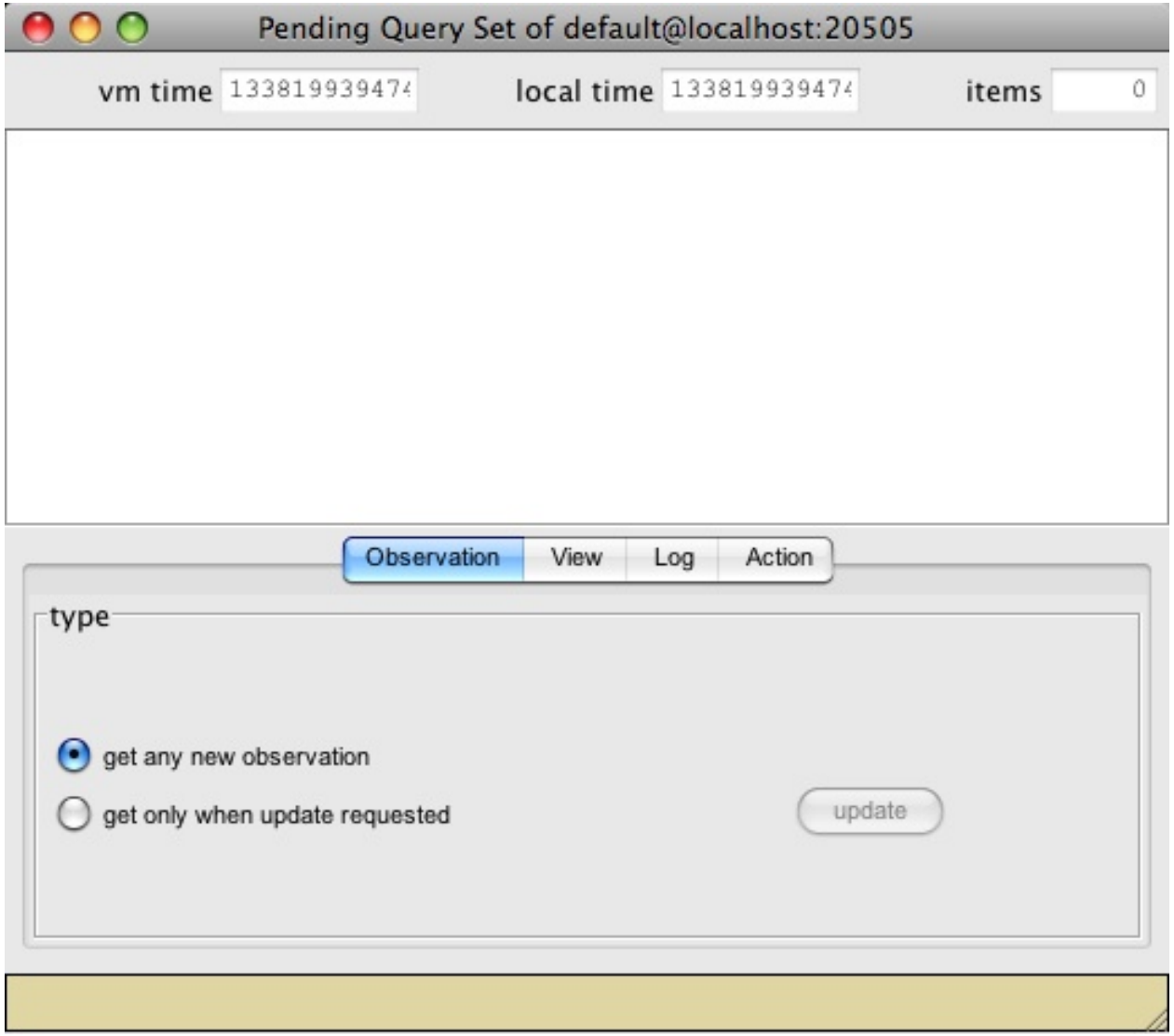
TuCSoN Tools III



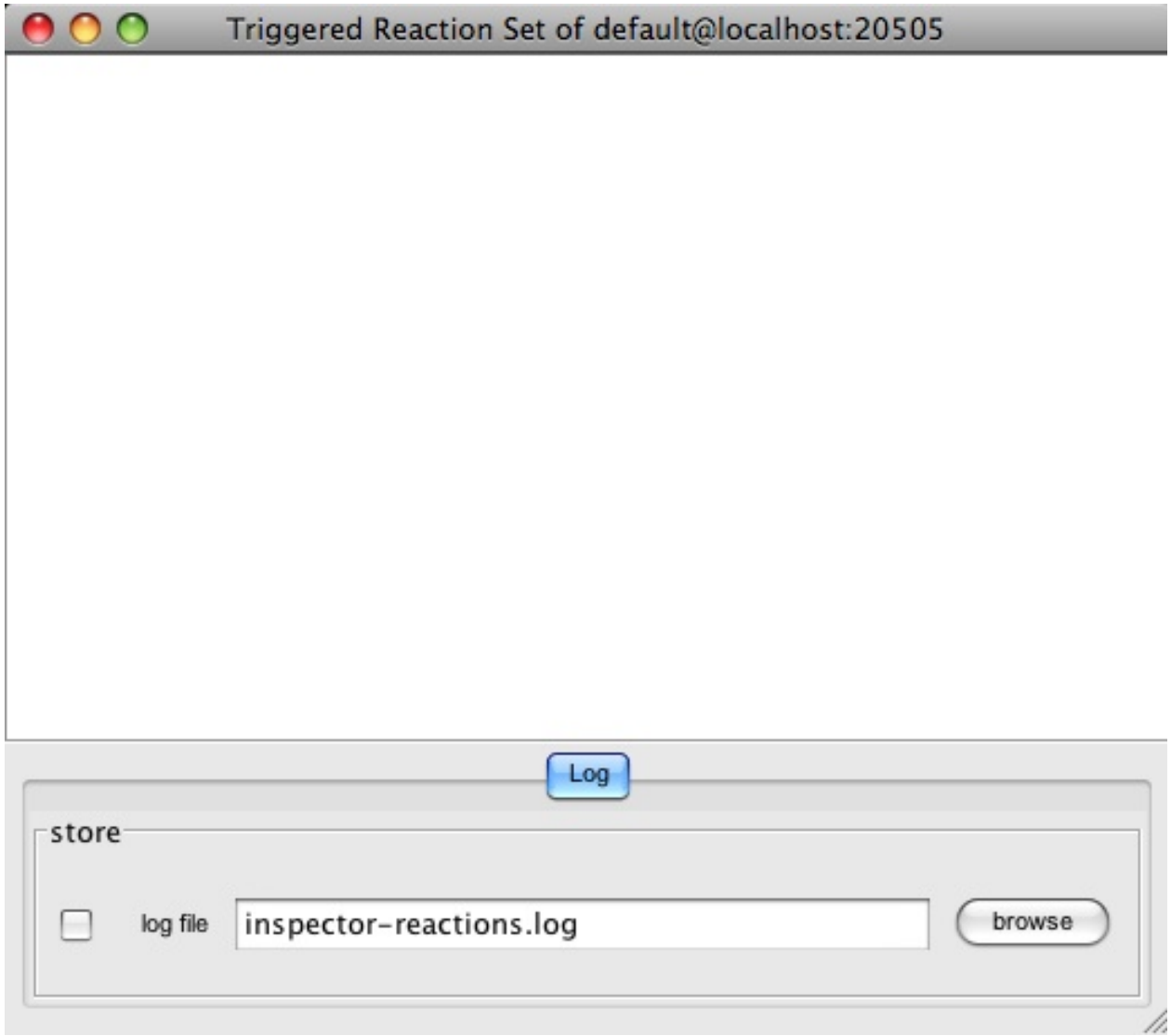
TuCSoN Tools IV



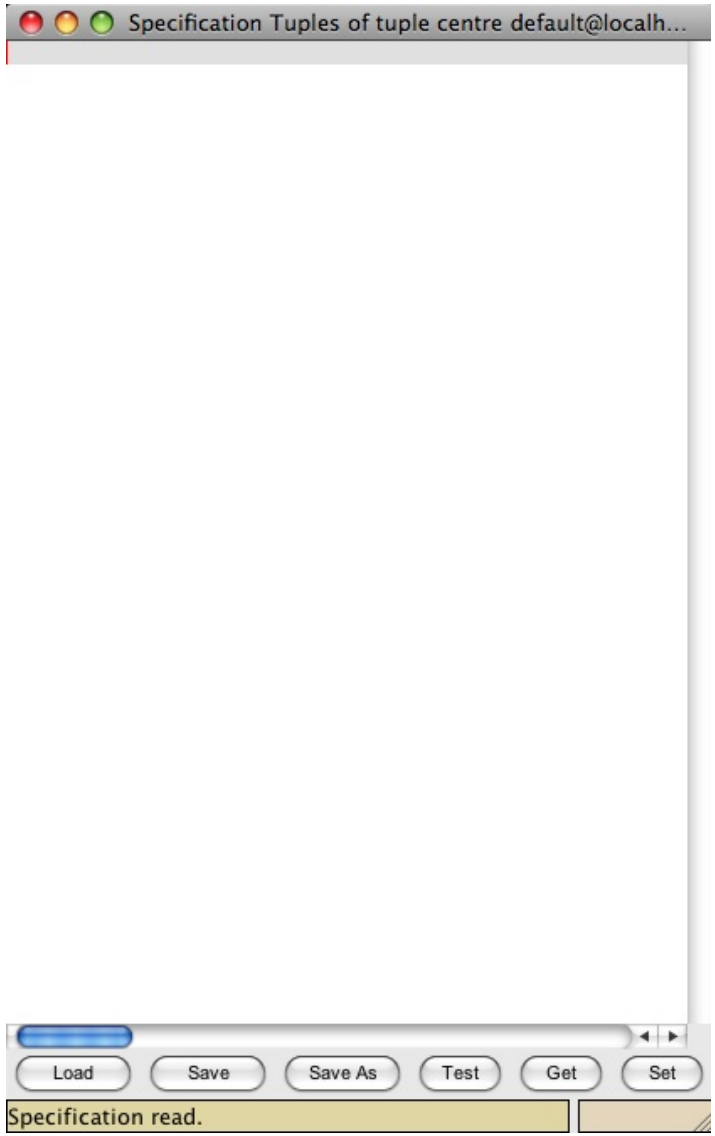
TuCSoN Tools V



TuCSoN Tools VI



TuCSoN Tools VII



Part 1: Basic TuCSoN

Experiments Page

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

HelloWorld.java | HelloAgent.java | HelloAgentTest.java | TucsonAgent.java

```
16
17 public class HelloWorld {
18
19     public static void main(String[] args) {
20
21         TucsonAgentId me = null;
22         TucsonTupleCentreId ttcid = null;
23         try {
24             me = new TucsonAgentId("helloAgent");
25             ttcid = new TucsonTupleCentreId("default", "localhost", "20504");
26         } catch (TucsonInvalidAgentIdException e1) {
27             e1.printStackTrace();
28         } catch (TucsonInvalidTupleCentreIdException e) {
29             e.printStackTrace();
30         }
31
32         SynchOnlyACC acc = TucsonMetaACC.getContext(me, "localhost", 20504);
33         long now = System.currentTimeMillis();
34         LogicTuple tuple = new LogicTuple("msg", new Value("Hello World!"), new Value("time", new Value(now)));
35         try {
36             acc.out(ttcid, tuple, null);
37             System.out.println("Tuple inserted: " + tuple);
38             LogicTuple template = new LogicTuple("msg", new Var("Msg"), new Var("Time"));
39             LogicTuple msg = acc.in(ttcid, template, null);
40             System.out.println("Tuple retrieved name: " + msg.getName());
41             System.out.println("Msg argument: " + msg.getArg(0));
42             System.out.println("Time argument: " + msg.getArg(1).getArg(0));
43             acc.exit();
44         } catch (TucsonOperationNotPossibleException e) {
45             e.printStackTrace();
46         } catch (UnreachableNodeException e) {
47             e.printStackTrace();
48         } catch (OperationTimeOutException e) {
49             e.printStackTrace();
50         } catch (InvalidTupleOperationException e) {
51             e.printStackTrace();
52         }
53
54     }
55
56 }
```

Example 3: Hello World from Java with TucsonAgent I

HelloWorld.java | HelloAgent.java | HelloAgentTest.java | TucsonAgent.java

```
15 public class HelloAgent extends TucsonAgent{
16
17     protected HelloAgent(String aid) throws TucsonInvalidAgentIdException {
18         super(aid);
19     }
20
21     @Override
22     protected void main() {
23
24         TucsonTupleCentreId ttcid = null;
25         try {
26             ttcid = new TucsonTupleCentreId("default", "localhost", "20504");
27         } catch (TucsonInvalidTupleCentreIdException e) {
28             e.printStackTrace();
29         }
30
31         SynchOnlyACC acc = getContext();
32         long now = System.currentTimeMillis();
33         try {
34             LogicTuple tuple = LogicTuple.parse("msg('Hello World!', time(" + now + "))");
35             acc.out(ttcid, tuple, null);
36             say("Tuple inserted: " + tuple);
37             LogicTuple template = LogicTuple.parse("msg(Msg, Time)");
38             LogicTuple msg = acc.in(ttcid, template, null);
39             say("Tuple retrieved name: " + msg.getName());
40             say("Msg argument: " + msg.getArg(0));
41             say("Time argument: " + msg.getArg(1).getArg(0));
42             acc.exit();
43         } catch (TucsonOperationNotPossibleException e) {
44             e.printStackTrace();
45         } catch (UnreachableNodeException e) {
46             e.printStackTrace();
47         } catch (OperationTimeOutException e) {
48             e.printStackTrace();
49         } catch (InvalidTupleOperationException e) {
50             e.printStackTrace();
51         } catch (InvalidLogicTupleException e) {
52             e.printStackTrace();
53         }
54
55     }
```

Example 3: Hello World from Java with TucsonAgent II

HelloWorld.java | HelloAgent.java | HelloAgentTest.java | TucsonAgent.java

```
1  package tucson.examples.hello_agent;
2
3  import alice.tucson.api.exceptions.TucsonInvalidAgentIdException;
4
5  public class HelloAgentTest {
6
7      /**
8       * @param args
9       */
10     public static void main(String[] args) {
11         try {
12             new HelloAgent("helloAgent").spawn();
13         } catch (TucsonInvalidAgentIdException e) {
14             e.printStackTrace();
15         }
16     }
17
18 }
19
```

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

TuCSoN Organisation I

RBAC

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

[a] 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 (`$ORG`) contains the dynamic rules of RBAC in TuCSoN

! the current TuCSoN implementation provides an unstable and unreliable implementation of RBAC

TuCSoN Agent Coordination Contexts I

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

TuCSoN Agent Coordination Contexts II

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

TuCSoN Agent Coordination Contexts III

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

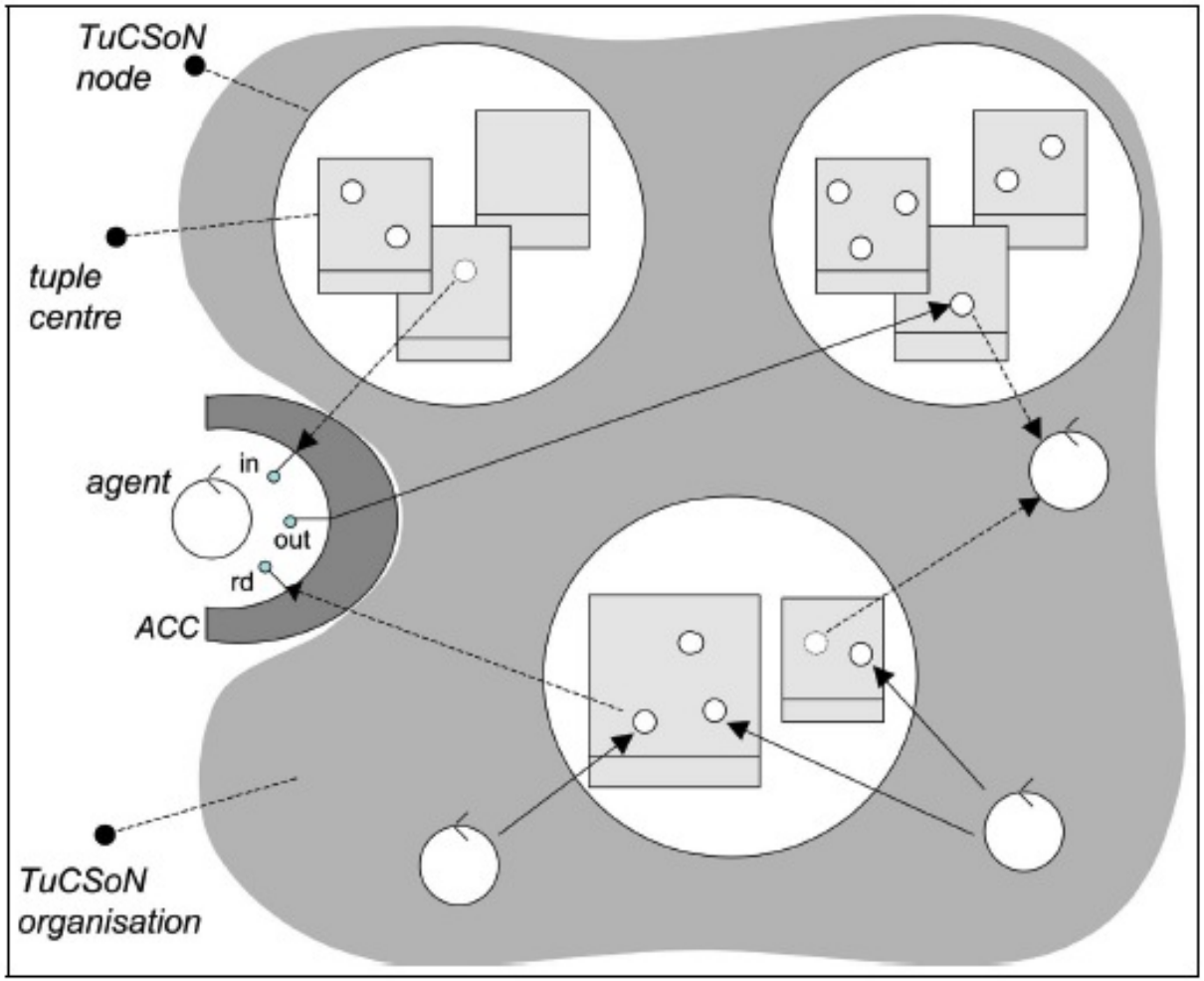
`SpecificationSynchACC` enables interaction with the specification space and enacts a blocking behaviour from the agent's perspective: whichever the meta-coordination operation invoked (either suspensive or predicative), the agent stub *blocks* waiting for its completion

`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

TuCSoN System



Part 2: Advanced TuCSoN

TuCSoN Coordination Language I

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*
- meta-coordination operations are invoked by agents upon tuple centres, which are then to be univocally referred in the operation

TuCSoN Coordination Language II

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 Coordination Language III

Coordination Primitives

- 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

`out_s(E,G,R)` writes a specification tuple `reaction(E,G,R)` in the target tuple centre—where `reaction(E,G,R)` belongs to the specification language

`rd_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 first served, the execution is suspended, to be resumed and completed when a matching `reaction(E,G,R)` specification tuple is finally found on the target tuple centre, and returned

`in_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 first served, the execution is suspended, to be resumed and completed when a matching `reaction(E,G,R)` 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

TuCSoN Meta-Coordination Operations III

Test-for-Absence Meta-Operation

`no_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 a matching `reaction(E,G,R)` specification tuple is found when the operation is served, the execution fails, and `reaction(E,G,R)` is returned; otherwise the operation succeeds

Space Meta-Operations

`get_s()` reads all the specification tuples in the target tuple centre, and returns them as a list

`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)`

Part 2: Advanced TuCSoN

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

Part 3: Conclusion

TuCSoN & Beyond I

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

TuCSoN & Beyond II

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

Bibliography I

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

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.

Bibliography II


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.


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.

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