



TuCSoN Guide

TuCSoN version: 1.4.5

Document revision: 003

Creation date: 2002-04-05

Last Changes date: 2006-11-04

<http://alice.unibo.it/xwiki/bin/view/TuCSoN/>

apiCE Lab, Università di Bologna, Cesena, Italy
DEIS, Università di Bologna, Italy

Contents

I	TuCSoN Basics	3
1	Introduction	4
2	Getting started	5
3	TuCSoN Tools	9
3.1	The CLI tool	9
3.2	The Inspector tool	10
4	TuCSoN API: accessing TuCSoN from Java and Prolog agents	17
4.1	Accessing TuCSoN from Java	17
4.1.1	Logic Data Structures	17
4.1.2	The Agent Coordination Context Class and Related	19
4.2	Accessing TuCSoN from tuProlog	21
II	TuCSoN Advanced	24
5	Programming Tuple Centre Behaviour	25
5.1	The ReSpecT Language	25
5.2	Coordination as computations on the interaction space	27
5.2.1	Basic computation constructs in ReSpecT: Selection, Iteration and Recursion . .	27
5.3	Common mistakes	29
5.3.1	Binding variables more than once	29
III	Engineering systems with TuCSoN	32
6	How to engineer systems: a Task-oriented approach	33
6.1	Task-oriented engineering	33
6.2	Design and development of coordination in TuCSoN	34
7	Examples and Patterns	35
7.1	Some Communication Patterns	35
7.1.1	Blackboard-based Communication	35
7.1.2	Message-oriented Communication	35
7.1.3	Remote Procedure Calls and Services	38
7.2	Simulating New Coordination Primitives	39
7.2.1	rd_all	40
7.2.2	in_all	40
7.3	Basic Synchronisation Patterns	41

7.3.1	Semaphores	41
7.3.2	Synchronisation Barrier	41
7.4	Other Basic Coordination Patterns	42
7.4.1	Interoperability	42
7.4.2	Resource Allocation	42
7.5	Some Coordination Idioms	42
7.5.1	Tuple on demand	43
7.6	Other examples	43
7.6.1	Dining Philosophers	43
7.6.2	Contract Net Protocol	43
7.6.3	Workflow Engine	43
IV	Extensions	44
8	Tuple centre persistency	45
8.1	Introduction	45
8.2	Enabling and disabling persistency	45
8.3	Persistent strategy	46
9	Timed ReSpecT	47
9.1	Introduction	47
9.2	Extending ReSpecT with Time	47
9.3	Examples	49
9.3.1	Timed Requests	49
9.3.2	Tuples in Leasing	51
9.3.3	Dining Philosophers with Maximum Eating Time	51
9.3.4	Dining Philosophers with Maximum Eating Time	51
9.3.5	An Artifact for Timed Contract Net Protocols	53
10	Extending TuCSoN with Organisation and Security Support: RBAC	56
A	Related Documents and Publications	57
B	Document History	58
C	Planned Development of this Document	59

Part I

TuCSoN Basics

Chapter 1

Introduction

TuCSon is an infrastructure providing services enabling the communication and coordination of distributed / concurrent independent software components called *agents*. In this context, any autonomous software component embedding its own control flow(s) can be considered an agent. As objects of the OOP paradigm, agents encapsulate a state and a behaviour; differently from objects, they have control of both. (objects has not control of their behaviour: sending a message to an object causes method invocation and execution). So agents do not interact by means of method invocation – which causes the control to flow from one object to an other – but through some interaction/coordination model, which can be based on message passing or shared communication/coordination abstractions such as blackboards, tuple spaces, etc. TuCSon infrastructure supports agent communication and coordination providing *tuple centres*, which are shared & reactive information spaces, distributed over the infrastructure nodes, where agents insert, retrieve, read information in the form of *tuple*, ordered collections of heterogeneous information chunks. More specifically, tuple centres are *programmable tuple spaces*, i.e. tuple spaces with a reactive behaviour which can be programmed dynamically (by humans as well as by agents). Agents access tuple centres associatively by writing, reading, and consuming tuples via simple communication operations (*out*, *rd*, *in*, *inp*, *rdp*).

Tuple space model has been introduced originally as powerful way for realising communication and coordination in concurrent systems, alternatively to message-based and shared-variable based models. Among the key points that make tuple space models different from these models is *generative communication* style: agents communicate by creating and retrieving associatively tuples whose existence – once created – is independent with respect to agents' one. This makes it possible to obtain uncoupling properties – temporal and spatial – which greatly simplify the engineering of the agent interaction space in systems' development.

Tuple centres specialise and extend the basic tuple space model by (1) using logic tuples: tuples are first order logic terms (such as Prolog terms); (2) while the behaviour of a tuple space in response to communication events is fixed, the behaviour of a tuple centre can be tailored to the application needs by defining a set of *specification tuples* expressed in the ReSpecT language, which define how a tuple centre should react to incoming/outgoing communication events [12]. So, differently from tuple spaces – and related implementation such as JavaSpaces [6] or T-Spaces [17] – tuple centres can be programmed with reactions so as to encapsulate coordination laws directly in the coordination media.

In order to exploit TuCSon services, both a set of API (Application Programming Interface) and a set of tools are provided. The former are meant to be exploited when programming agents, to be used directly in the agent program to interact with tuple centres. Currently API in Java (packages *alice.tucson.api* and *alice.logictuple*) and for tuProlog agents are provided. The latter instead are meant to be exploited by humans to interact and inspect/debug tuple centres, Command Line Interpreter (CLI) and Inspector tool, respectively.

Chapter 2

Getting started

The infrastructure is set up by installing (at least) one node. Once you have properly installed TuCSoN in the *dir* directory, type:

```
java -cp dir/tucson.jar alice.tucson.service.Node
```

In the case you uncompressed the jar file, or installed the complete installation including Java sources, you need not specify the jar file: switch to the installation directory and issue the following command at a command prompt:

```
java alice.tucson.service.Node
```

To test if the service is active on a specific host (in this case local host) you can use the `telnet` application:

```
telnet localhost 20504
```

20504 is the default port where is located the service. Of course you can install the infrastructure on a different port (specifying the port as an argument of for the `tucson.service.Node` application). If the service is active (which means that the host is a TuCSoN node), after the `telnet` invocation it should appear a message like this:

```
TuCSoN Infrastructure (version X.X.X)
Current date YYY
Running since ZZZ
```

The infrastructure can be installed also on a different port, by specifying the port value with the `-port` option:

```
java -cp dir/tucson.jar alice.tucson.service.Node -port 20000
```

In this example TuCSoN is installed on port 20000.

Then you are ready to interact with the infrastructure, inserting and retrieving tuple from tuple centres of that node and changing their behaviour. The quickest way to interact with is through the *Command Line Interpreters*, which provide a simple interface to invoke directly tuple centre coordination language primitives (actions) (`out,in,rd,inp,rdp set_spec` and `get_spec`). So from a console command type:

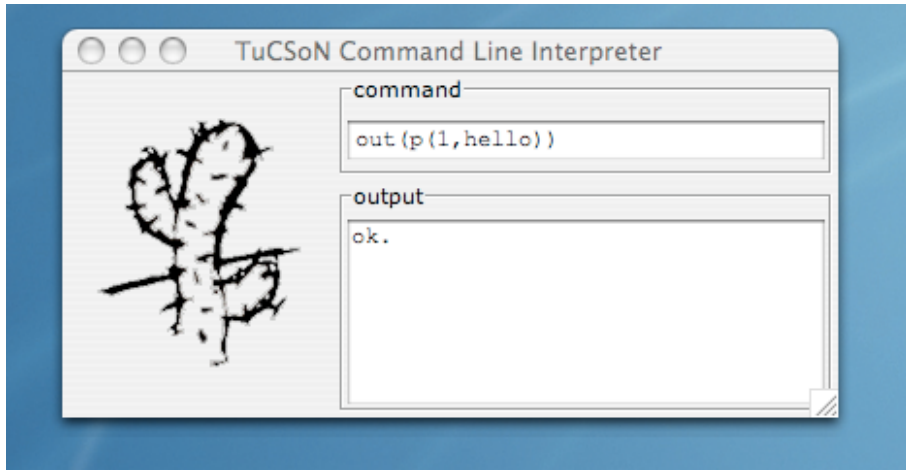


Figure 2.1: CLI-Agent tool

```
java -cp dir/tucson.jar alice.tucson.tools.CLI
```

You should see the GUI represented in Figure 2.1 (by specifying the `-console` option, you can use the console version of the command line interpreter). In the command edit of the windows of CLI, type (with carriage return):

```
out(p(1,hello))
```

you are asking the TuCSoN infrastructure to put the tuple `p(1,hello)` in the default tuple centre; if everything goes well you should see the `ok` message in the output window. Now we want first read the tuple and then retrieve it; first, type the command:

```
rd(p(X,Y))
```

with this command, you are asking for a tuple having name `p` and 2 arguments; `X` and `Y` are the variables that will be bound to actual values, in the case a matching is found. Since you are the only user of the TuCSoN infrastructure, the answer to the issued command should appear:

```
answer: p(1,hello)
```

Now let's retrieve (not only read) the tuple; so type the command:

```
in(p(_,hello))
```

In this case a different tuple template is used: we are saying that we want to retrieve a tuple named `p`, ignoring its first argument and whose second argument is the atom `hello`. The answer would be the same as before, but with this command we have taken away the tuple from the tuple centre.

Now let's test the blocking behaviour of tuple centres. Type a request for a tuple not actually present in the default tuple centre:

```
in(researcher(id(13),Surname,Name))
```

As you can see, the CLI has blocked, since there isn't at the moment a tuple matching the re-

quest and waiting for it. So open a new CLI and type the command:

```
out(researcher(id(_),'Bukowski','Charles'))
```

you should see the blocked CLI to be unblocked, reporting the answer:

```
answer: researcher(id(13),'Bukowski','Charles')
```

Now we want to change the behaviour of the tuple centre by means of `set_spec` coordination primitive - remember in fact that tuple centres are programmable coordination media and the programming language used is the ReSpecT language. So always using the CLI, type the following command:

```
set_spec('reaction(out(X),( out_r(backup(X))))'.')
```

If the new tuple centre specification has been accepted, you should receive as answer the specification itself; otherwise if you receive an `answer: no.` it means that there was an error in the specification. So, let's suppose that the specification has been accepted: we have programmed the tuple centre in order to store a backup tuple every time a tuple is inserted in centre. To test it, type any `out` command from the CLI:

```
out(array(index(1),value(tomato)))
```

Then, to test the effectiveness of the tuple centre behaviour, type the request:

```
rd(backup(X))
```

you should receive immediately the answer:

```
answer: backup(array(index(1),value(tomato))).
```

that is: when you inserted the tuple `array(index(1),value(tomato))`, the reaction described in the specification has been triggered and it produced a new tuple, `backup(array(index(1),value(tomato)))`.

Until now, we have interacted with the default tuple centre; now let's interact also with other tuple centres, specifying their names in the command invocation, as provided by the TuCSoN coordination model. Always by means of the CLI just type:

```
temperature ? out(data(jan,[15,17.5,14]))
```

The effect of this command is to place the tuple `data(jan,[15,17.5,14])` in the tuple centre `temperature`. Then we retrieve the tuple by means of the `inp` not blocking invocation:

```
temperature ? inp(data(jan,X))
```

If a coordination primitive refers to a not existent tuple centre, a new tuple centre with specified name is created.

TuCSoN is coordination infrastructure for Internet environment, so fully enjoyable in particular using it in a network; in particular the coordination primitives can be used to access tuple centres of

remote nodes, identifying them by means of their complete identifier:

<Tuple Centre Name> @ <Tuple Centre IP Address>

The identifier are represented by logic terms, in particular Prolog term, so Prolog grammar rules apply: apices are necessary for strings containing dots, spaces, and special characters. Also, they are necessary for strings starting with an upper case, in order to distinguish them from variables.

Some examples:

```
temperature @ '137.204.192.175'
default @ 'deis.unibo.it'
folder(5) @ 'tucson.alice.unibo.it'
Archive @ localhost
```

So let's suppose that you are in network. First of all: how can you know if a specific host provides the TuCSoN service? As already seen, we can use the `telnet` application - suppose for instance that the host name is `acme.org`:

```
telnet acme.org 20504
```

Suppose that the service is present (or that you have access to that host and setup the infrastructure as seen at the beginning); then you can interact with tuple centres located at `acme.org` in the same way we have done previously, with the only difference that you have to fully specify (so name and address) the tuple centre identifiers. For instance (always using a CLI):

```
temperature @ 'acme.org' ? out(data(jan,[15,17.5,14]))
```

Then, for instance:

```
temperature @ 'acme.org' ? rdp(data(X,Y))
```

This (non blocking) command checks the presence of a matching tuple and reads it.

Chapter 3

TuCSoN Tools

3.1 The CLI tool

The TuCSoN CLI tool allows users to invoke the commands of the TuCSoN coordination language, which are of the general form:

$$TCName @ TCAAddress ? op(Tuple)$$

where:

- *TCName* is the tuple centre name: any string may be used, including strings with spaces (in that case enclosed by apices). So, `room`, `'Multimedia'`, `team05`, `'my documents'`, for instance, are all valid tuple centre names.
- *TCAAddress* is the tuple centre address: it must be a valid host name or IP address — represented as logic term —, such as `'deis.unibo.it'`, `'192.0.0.3'`, or `myHost`.
- *op* is the operation to be performed: see Table 3.1 for a short reference on allowed operations.
- *Tuple* is the information content involved in the operation: it must be a valid logic tuple or tuple template (that is, roughly speaking, a Prolog term). Examples include `pippo(1)`, `authors(['Bukowski', 'Joyce'])`, `pippo(X)`, `writer(name('Charles'), surname('Bukowski'))`, etc.

A key feature of the TuCSoN model is to provide a twofold view of the coordination space, which can be seen both as a *global resource* and as a *local resource*. As a global resource, a tuple centre is available from anywhere on the network, by explicitly specifying its name and address, as above. As a local resource, instead, a tuple centre is referenced simply by name, since the implicit address is the address of the local node. In this case, operation commands assume the simplified form:

$$TCName ? op(Tuple)$$

Moreover, in order to provide an easy-to-use support for simple coordination needs, TuCSoN defines a *default tuple centre* for each node, which is called `default`: if the tuple centre name is omitted, like in the case below, the default tuple centre is assumed:

$$op(Tuple)$$

Let us see some examples:

- `table @ 'deis.unibo.it' ? out(chops(1,2))`
Outputs tuple `chops(1,2)` into the tuple centre `table` of node `deis.unibo.it`.

- `docs ? out(article(title('TuCSoN'),authors(['Bukowski','Joyce'])))`
Outputs tuple `article(title(...),authors(...))` into tuple centre `docs` of the local node.
- `'my docs' ? rdp(p(First,Second))`
Checks for the presence of a tuple matching the template `p(First,Second)` in tuple centre `my docs` of the local node.
- `docs @ '192.0.0.3' ? in(article(title(X),authors([Y,'Joyce'])))`
Removes a tuple matching the template `article(title(...),authors(...))` from tuple centre `docs` of the node denoted by the IP address 192.0.0.3.
- `set_spec('reaction(in(p(X)), (pre, out_r(p(1))))')`
Adds the specification tuple `reaction(...)` to the default tuple centre of the local node.
- `lab00 @ '137.204.107.188' ? set_spec('reaction(out(X),(out_r(copy(X))))')`
Adds the specification tuple `reaction(...)` to the tuple centre `lab00` of the node denoted by the IP address 137.204.107.188.
- `default @ localhost ? get_spec(Spec)`
Gets a specification tuple from the default tuple centre of the local node.
- `in(p(X,1))`
Removes a tuple matching the template `p(X,1)` from the default tuple centre of the local node.

The command `quit` can be used to exit the interpreter.

3.2 The Inspector tool

The **Inspector** tool allows users to observe and debug the communication state and the behaviour of a tuple centre. In particular, it makes it possible to inspect the *tuple set*, the *pending query set*, the *triggered reaction set* and the *behaviour specification set*. Inspector also supports step-by-step tracing.

An inspector can be spawn in two ways, either by means of the *Inspector* application without specifying any argument: the application provides a GUI to specify the name and the address of the tuple centre to be inspected. Or it can be spawn providing all such information directly at the command prompt. In the first case, the command line look like the following (and the result is in Figure 3.1) :

```
java -cp tucson.jar alice.tucson.tools.Inspector
```

In the second case, instead, the tuple centre name and address must be specified:

```
java -cp tucson.jar alice.tucson.tools.Inspector TCName TCAAddress
```

For instance,

```
java -cp tucson.jar alice.tucson.tools.Inspector
myDocs tucson.alice.unibo.it
```

launches an Inspector on tuple centre `myDocs` at node `tucson.alice.unibo.it`

```
java -cp tucson.jar alice.tucson.tools.Inspector
lab00 137.204.107.188
```

out	to put the specified tuple in the specified tuple centre
in	to remove a tuple matching the specified tuple template from the specified tuple centre
rd	to read a tuple matching the specified tuple template from the specified tuple centre
inp	to remove, if present, a tuple matching the specified tuple template from the specified tuple centre (if not present the operation fails)
rdp	to read, if present, a tuple matching the specified tuple template from the specified tuple centre (if not present the operation fails)
set_spec	to set the behaviour of the specified tuple centre; tuple must be a valid ReSpecT program, in the form of a text (understood as a string atom) or list of reaction
get_spec	to get current behaviour of the specified tuple centre

Table 3.1: TuCSoN operations

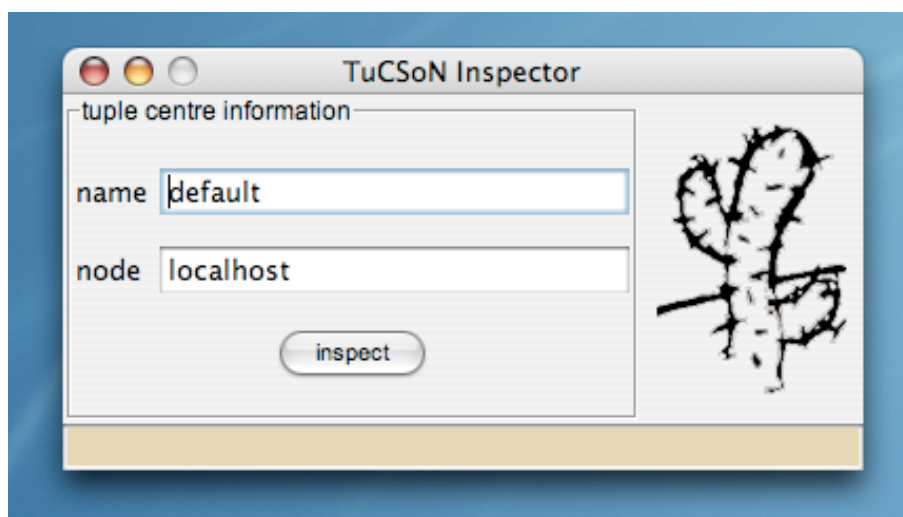


Figure 3.1: The Inspector GUI

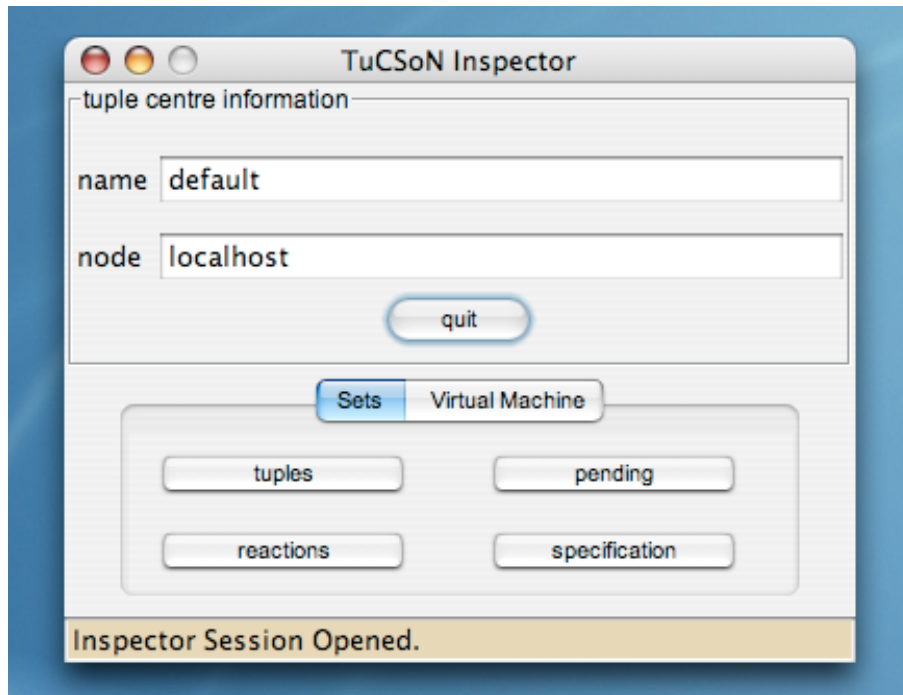


Figure 3.2: An Inspector operating on the `default@localhost` tuple centre

launches an Inspector on tuple centre `lab00` at the node whose IP address is 137.204.107.188. If *TCAddress* is omitted, `localhost` is assumed by default; if both arguments are missing, the default tuple centre at localhost is assumed.

In Figure 3.2 an inspector on default tuple centre at localhost has been spawned.

You can act on two different components of a tuple centre: the tuple centre sets and the tuple centre virtual machine, which are identified by the two different tabbed panels. Inspectable sets are:

- **Tuple set** – pressing the button `tuples` you start inspecting the tuple set of the tuple centre on a separate window (Figure 3.3). *vm time* is the own time of the inspected tuple centre virtual machine (in milliseconds from 1 Jan 1970) related to the last observation perceived by the inspector. *local time* is the own time of the inspector (again in milliseconds from 1 Jan 1970) related to the last perceived observation. In the Observation tab pane, you can choose the type of observation you want to apply: you can have a continuous observation, which means that the tuple set view is updated anytime the set is accessed; or you can choose to receive updated view of the tuple set only when requested (by pushing the `update` button).

In the View tab pane, you can select a filter in order to display only a subset of tuples; in particular, by specifying a tuple template, you enforce the visualization of only those tuples matching the template.

Through the Log tab pane you can enable the logging of the tuple set on a specific file, also providing filtering of the tuples always by means of a tuple template.

Through the Action tab pane you can enforce a specific set of tuples in one shot; in fact the text area where the tuples are displayed is editable, so you can change or add or remove tuples, creating a new tuple set description and try to enforce it. The new tuple set is accepted only if there aren't syntax errors in the new description.

- **Pending Query set** – pressing the button `pending` you start inspecting on a separate window (Figure 3.4) the set of the pending queries, that is the `in` and `rd` invocations that currently are

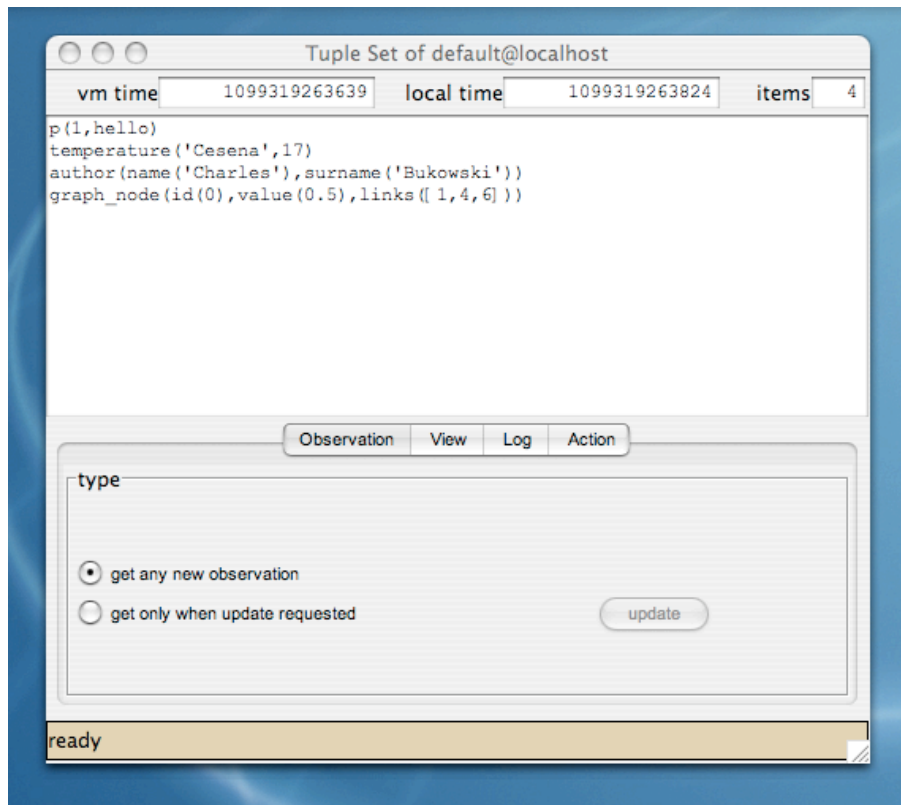


Figure 3.3: The Tuple Set View

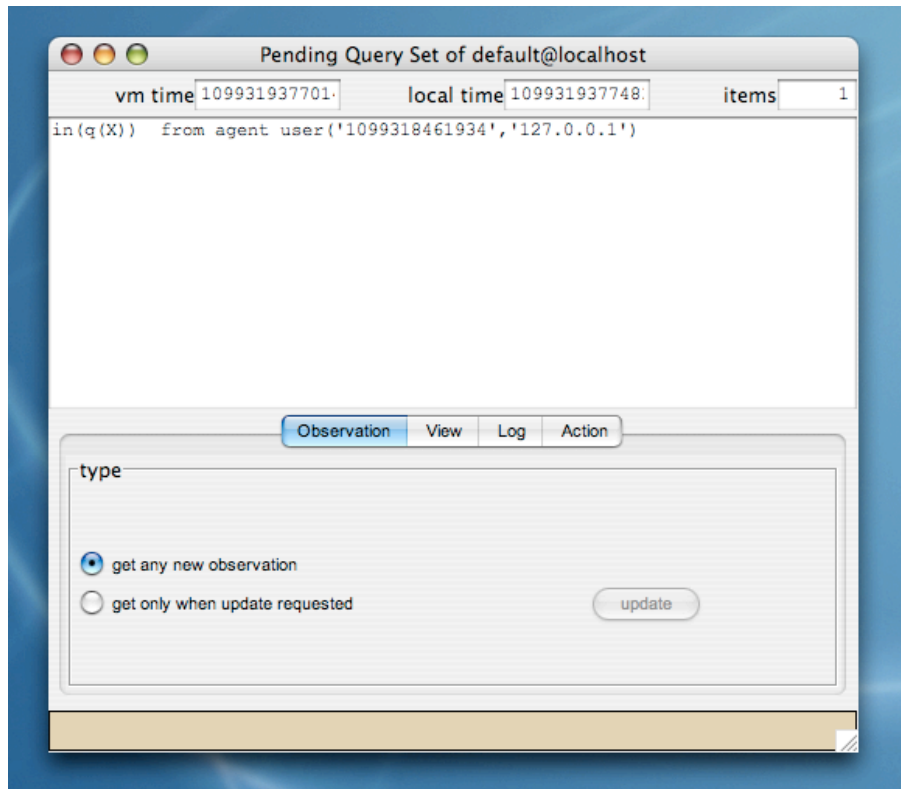


Figure 3.4: The Pending Query Set View

not satisfiable. The pending queries are represented as **query** tuples, whose first field represents the agent invocation (**in** or **rd**) and the second represents the identifier of the agent making the request. As already seen, **vm time** is the own time of the inspected tuple centre virtual machine (in milliseconds from 1 Jan 1970) related to the last observation perceived by the inspector. **local time** is the own time of the inspector (again in milliseconds from 1 Jan 1970) related to the last perceived observation. In the Observation tab pane, you can choose the type of observation you want to apply: you can have a continuous observation, which means that the pending query set view is updated anytime the set is accessed; or you can choose to receive updated view of the pending query set only when requested (by pushing the **update** button).

In the View tab pane, you can select a filter in order to display only a subset of pending queries; in particular, by specifying a tuple template, you enforce the visualization of only those tuples (representing queries) matching the template.

Through the Log tab pane you can enable the logging of the pending query set on a specific file, also providing filtering of the tuples always by means of a tuple template.

Through the Action tab pane you can enforce a specific set of queries in one shot; in fact the text area where the tuples representing the pending queries are displayed is editable, so you can change or add or remove tuples, creating a new tuple set description and try to enforce it. The new pending query set is accepted only if there aren't syntax errors in the new description.

- **Reaction set** – pressing the button **reactions** you start inspecting the set of the reactions that are triggered during the tuple centre computation (Figure 3.5). As you can see in the figure, both success and failed reactions are displayed, with the virtual machine time in which they were triggered.

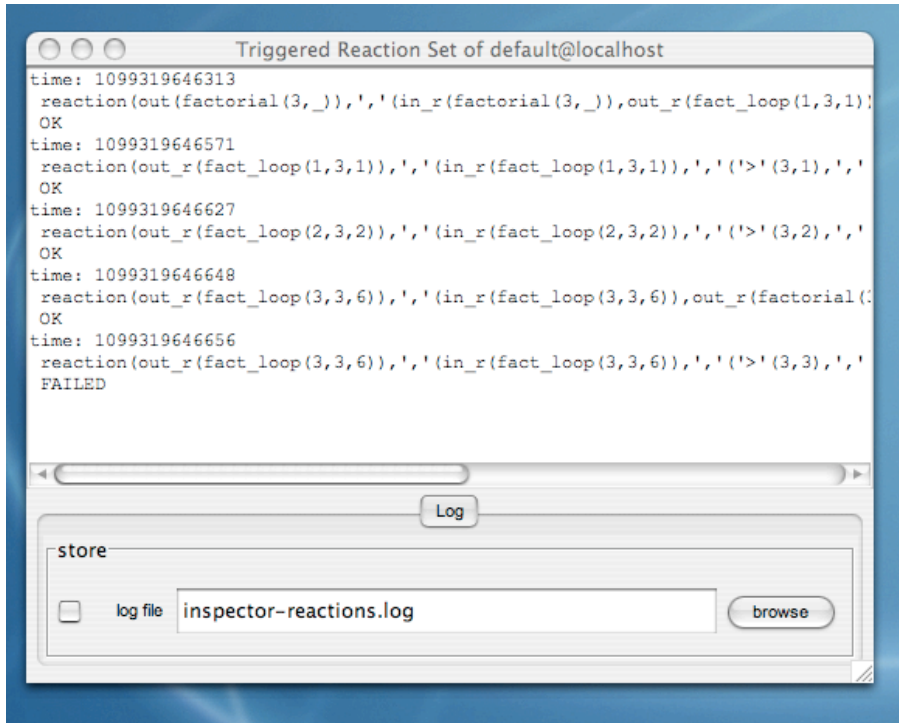


Figure 3.5: The Reaction Set View

Through the Log tab pane you can enable the logging of the triggered reactions on a specific file.

- **Specification set** – pressing the button **specification** you start inspecting and editing the behaviour of the tuple centre, characterized by a set of tuples representing the reaction specification (Figure 3.6).

Reactions are displayed with some form of context highlighting, with a different displaying style for coordination primitives (**in**,**out**,**rd**,**inp**,**rdp**) and ReSpecT primitives.

Some commands are provided (by means of the buttons at the bottom of the view) respectively to load, save, save as the ReSpecT specification on a file; the last two commands (the last two buttons) are fundamental, providing a way respectively to test the correctness of the specification and to send the new specification to the tuple centre.

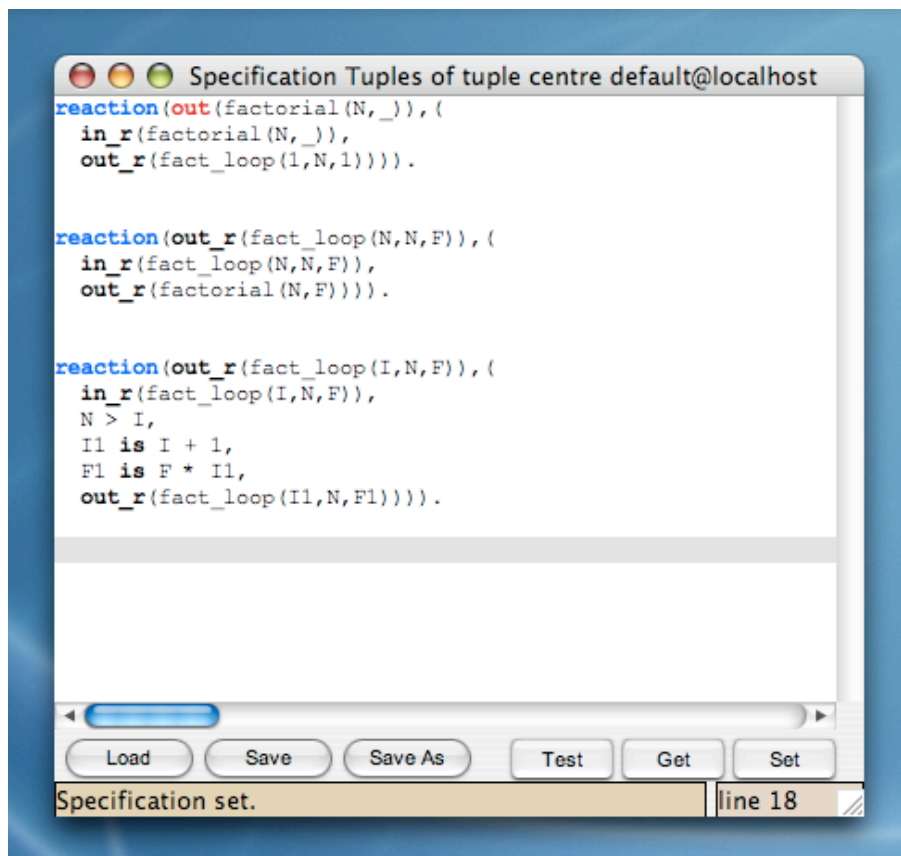


Figure 3.6: The Specification Set View

Chapter 4

TuCSoN API: accessing TuCSoN from Java and Prolog agents

TuCSoN has an application programming interface, which allows to exploit the service from applications written in Java and in `tuProlog` (our Prolog environment). In order to access the TuCSoN infrastructure, an *agent coordination context* (ACC) [10] must be first acquired. An ACC represents a sort of interface that an agent must have in order to invoke coordination primitives on tuple centres belonging to an organisation, living on TuCSoN. In this version of TuCSoN an ACC – represented by an object implementing the `TucsonContext` interface – can be obtained by calling one of the `getContext` static method of the class `Tucson`. The method results in an ACC object which the agent can exploit to interact with tuple centres.

4.1 Accessing TuCSoN from Java

The fundamental packages, required by any Java program wishing to exploit the TuCSoN infrastructure, are `alice.tucson.api` and `alice.logictuple`. `alice.tucson.api` package includes classes providing access to TuCSoN infrastructure, classes representing tuple centre identifier (`TupleCentreId`) and agent identifier (`AgentId`). `alice.logictuple` package include classes representing the communication language of TuCSoN, that is logic tuple.

4.1.1 Logic Data Structures

The main classes that constitute the communication language are:

- **LogicTuple** – this class represents a logic tuple. Typically a logic tuple is characterised by a name and a list of logic arguments (**TupleArgument**);
- **TupleArgument** – this class represents the a tuple argument. It can be a value – numbers, strings, structures, lists – or a variable. This class provides all the services to manipulate tuple arguments, testing their type, converting them, retrieving their numeric / alphanumeric value. Derived classes **Value** and **Var** are provided for facilitating the construction of specific tuple arguments;
- **Value** – this class represents concrete data values of a logic argument. In particular **Value** objects can be created representing untyped values – such as atoms (`Strings`) and structures (with a name and a set of arguments) – and typed values such as numbers – integers, long, floats, doubles.
- **Var** – this class represents logic variable, characterised by a name.

Actually, classes **Value** and **Var** act only as specific constructors of tuple arguments: all the services for inspecting, manipulating, converting tuple arguments are provided by **TupleArgument** class.

The main methods provided by the `alice.logictuple.LogicTuple` class are the followings:

```

public class LogicTuple implements alice.tuplecentre.Tuple,
                                alice.tuplecentre.TupleTemplate,
                                java.io.Serializable {
    public LogicTuple(String name);
    public LogicTuple(String name, TupleArgument t1);
    public LogicTuple(String name, TupleArgument t1, TupleArgument t2);
    ...
    public LogicTuple(String name, TupleArgument t1, TupleArgument t2, TupleArgument t3,
                      TupleArgument t4, TupleArgument t5, TupleArgument t6,
                      TupleArgument t7);
    public LogicTuple(String name, TupleArgument[] list);

    public int getArity() throws InvalidTupleOperationException;
    public String getName() throws InvalidTupleOperationException;
    public TupleArgument getArg(int index) throws InvalidTupleOperationException;
    public TupleArgument getVarValue(String varName);

    public boolean match(alice.tuplecentre.Tuple t);
    public boolean propagate(alice.tuplecentre.Tuple t);

    public static LogicTuple parse(String st)
        throws InvalidLogicTupleException;
}

```

Instead the main methods provided by the `alice.logictuple.TupleArgument` class are the followings:

```

public class TupleArgument implements java.io.Serializable {

    public TupleArgument(){...};
    public TupleArgument(Term t){...};

    public boolean isVar() {...}
    public boolean isValue(){...}
    public boolean isAtom() {...}
    public boolean isNumber(){...}
    public boolean isInteger(){...}
    public boolean isReal(){...}
    public boolean isInt(){...}
    public boolean isFloat(){...}
    public boolean isLong(){...}
    public boolean isDouble(){...}
    public boolean isStruct(){...}
    public boolean isList(){...}
    public boolean isAtomic(){...}

    public int intValue() throws InvalidTupleOperationException {...}
    public long longValue() throws InvalidTupleOperationException {...}
    public float floatValue() throws InvalidTupleOperationException {...}
    public double doubleValue() throws InvalidTupleOperationException {...}
    public List toList() throws InvalidTupleOperationException {...}
    public TupleArgument[] toArray() throws InvalidTupleOperationException {...}
    public alice.tuprolog.Term toTerm(){...}
    public String toString(){...}

    public String getName() throws InvalidTupleOperationException {...}
    public TupleArgument getArg(int index) throws InvalidTupleOperationException {...}
    public TupleArgument getVarValue(String varName) {...}
    private Term getVarValue(String name, Struct st){...}
    public int getArity() throws InvalidTupleOperationException {...}
    public java.util.Iterator listIterator(){...}

    public static TupleArgument parse(String st) throws InvalidTupleArgumentException {...}
}
%

```

Examples of logic tuple creation are:

```

%
LogicTuple simpleTuple = new LogicTuple("p", new Value(1));

```

```

LogicTuple simpleTuple2 = new LogicTuple("temperature",
    new Value("Rome"), new Value(17.5));

LogicTuple complexTuple =
    new LogicTuple("printer",
        new Value("my-printer"),
        new Value("type", new Value("laser"), new Value(300)),
        new Value("address", new Value("137.204.191.189")),
        new Value(0.70));

LogicTuple complexTuple2 =
    new LogicTuple("matrix",
        new Var("X"), new Var("X"), new Value(1));

LogicTuple simpleTupleTemplate = new LogicTuple("p", new Var());

LogicTuple complexTupleTemplate =
    new LogicTuple("printer",
        new Value("my-printer"),
        new Var("type", new Value("laser"), new Var()),
        new Value("address", new Var("X")),
        new Var());

LogicTuple tupleWithList =
    new LogicTuple("temperatures",
        new TupleArgument[]{ new Value(17.5), new Value(20), new Value(22.2) });

LogicTuple tupleWithListAsBefore = LogicTuple.parse("temperatures([17.5, 20, 22.2])");

LogicTuple complexTupleTemplate2 =
    new LogicTuple("matrix",
        new Value(1), new Value(1), new Var());

LogicTuple complexTuple3 =
    LogicTuple.parse(
        "printer('my-printer', type(laser, 300), address('137.204.191.189'), 0.70)");

LogicTuple complexTuple4 =
    LogicTuple.parse(
        "email_list(['aricci@deis.unibo.it', 'aomicini@deis.unibo.it', 'edenti@deis.unibo.it'])");

```

Points that are worth noting:

- the possibility to get a logic tuple from its text representation, by means of the static service `parse` method; in this case, the logic tuple syntax follows the standard first-order logic (Prolog) one, so in the case string with special symbols apices must be used. Note that also list can be specified;
- the method `getVarValue` makes it possible to get the value bound to a variable eventually present inside the logic tuple, directly by specifying the name of the variable. This method works only after that the logic tuple has been resolved, that is the tuple has been subject of matching or unification with an other tuple: typically, this method is used with a tuple used in or retrieved by a coordination primitive (which will be explained in next subsections).
- the method `toArray`, `toList`, `listIterator` make is possible to manage tuple arguments that are (Prolog) lists as they were Java array and lists, making easy the access to the list elements.

4.1.2 The Agent Coordination Context Class and Related

The ACC is represented by the `TucsonContext` interface, providing the basic coordination primitives already described in section 3.1. In TuCSoN 1.4.5 an ACC can be accessed only by one thread of control: so agents composed by multiple threads must obtain multiple ACCs.

```

package alice.tucson.api;
import alice.logictuple.*;

public interface TucsonContext {

    void out(TupleCentreId tid, LogicTuple t)
        throws OperationNotAllowedException,
               UnreachableNodeException;

    LogicTuple in(TupleCentreId tid, LogicTuple t)
        throws OperationNotAllowedException,
               UnreachableNodeException;

    LogicTuple rd(TupleCentreId tid, LogicTuple t)
        throws OperationNotAllowedException,
               UnreachableNodeException;

    LogicTuple inp(TupleCentreId tid, LogicTuple t)
        throws OperationNotAllowedException,
               UnreachableNodeException;

    LogicTuple rdp(TupleCentreId tid, LogicTuple t)
        throws OperationNotAllowedException,
               UnreachableNodeException;

    void setSpec(TupleCentreId tid, String spec)
        throws OperationNotAllowedException,
               UnreachableNodeException,
               InvalidSpecificationException;

    String getSpec(TupleCentreId tid)
        throws OperationNotAllowedException,
               UnreachableNodeException;

    void exit()
        throws OperationNotAllowedException;
}

```

Some points are worth noting:

- the `OperationNotAllowedException` exception, that is thrown if the performed act is not allowed according to the ACC;
- the `exit` method, used to release (exit) the acquired ACC.
- the logic tuple returned by the `in`, `inp`, `rd`, `rdp` coordination primitives refers to the logic tuple that has been provided as argument, after having been subject to the propagation (unification) computation involved by the primitive. These coordination primitives act directly on the logic tuples provided as (tuple template) argument, exploiting the propagation (unification) service that they provide.

An ACC description is represented by the class `ContextDescription`.

```

import alice.tucson.api.*;

public class ContextDescription implements java.io.Serializable {

    public ContextDescription(String theory);

    public ContextDescription(java.util.Properties p);

    ...
}

```

Table 4.1: An example of default agent coordination context usage

```
import alice.tucson.api.*;
import alice.logictuple.*;

public class Test {
    public static void main(String[] args) throws Exception {
        TucsonContextSynch cn =
            new TucsonContextSynch(Tucson.getContext(new AgentId("pippo")));
        TupleCentreId tid = new TupleCentreId("test.tc");
        cn.out(tid, new LogicTuple("p", new Value("hello world")));
        LogicTuple t = cn.in(tid, new LogicTuple("p", new Var("X")));
        System.out.println(t);
    }
}
```

The meta coordination context is represented by the **Tucson** class, providing the basic static services to negotiate/acquire an agent coordination context. Currently, this class acts as the mean to *discover* the infrastructure.

```
package alice.tucson.api;
import alice.logictuple.*;

public class Tucson {

    public static TucsonContext getContext(AgentId id)
        throws ContextNotAvailableException;

    public static boolean isActive(InetAddress node);

    public static String getVersion();
}
```

The easiest way to exploit the TuCSoN infrastructure is to exploit the class **TucsonContextSynch** by which agents can exploit an ACC object to interact with tuple centres. The interaction is synchronous. A little example is shown in Table 4.1. A more involved example concerns negotiating a default context is shown in Table 4.2.

Some helper classes are provided as root classes for building simple TuCSoN agents – as object with their own thread of control, able to access TuCSoN infrastructure, possibly not providing public methods and fields: **alice.tucson.api.TucsonAgent** class and **alice.tucson.api.Automaton** class. In particular, the last one is suitable to build agents in terms of automaton behaviour, with an explicit notion of state representable as a public method. Through the **become** method, the automaton can change its state, specifying the name of the next one, which must correspond to the name of a public method. Table 4.3 shows the same example as above, using **tucson** agents.

4.2 Accessing TuCSoN from tuProlog

In order to exploit TuCSoN from tuProlog, the library

alice.tucson.api.Tucson2PLibrary

must be loaded. The role of meta coordination context is provided a built-in predicate `enter_context/1`, which can be used to get an agent coordination context; the context description is provided as list of facts/clauses. Actually, in this version of TuCSoN in the context description can be specified only the agent identifier, with `agent_id(ID)`. For instance:

Table 4.2: An more articulated example about TuCSoN access through a default agent coordination context

```

import logictuple.*;
import tucson.api.*;

public class Test2 {
    public static void main (String args[]) throws Exception{

        AgentId aid=new AgentId("agent.0");
        TucsonContextSynch cn = TucsonContextSynch(Tucson.getContext(aid));

        // put the tuple value(1,38.5) on the temperature tuple centre
        TupleCentreId tid=new TupleCentreId("temperature");
        LogicTuple outTuple=LogicTuple.parse("value(1,38.5)");
        cn.out(tid,outTuple);

        // retrieve the tuple using value(1,X) as a template
        LogicTuple tupleTemplate=new LogicTuple("value",
                                                new Value(1),
                                                new Var("X"));
        LogicTuple inTuple=cn.in(tid,tupleTemplate);

        cn.exit();
        System.out.println(inTuple);
    }
}

```

Table 4.3: An example of `alice.tucson.api.TucsonAgent` usage

```

import alice.tucson.api.*;
import alice.logictuple.*;

class MyAgent extends TucsonAgent {
    public MyAgent(AgentId id){
        super(id);
    }

    protected void main(){
        try {
            TupleCentreId tid = new TupleCentreId("test_tc");
            TucsonContextSynch cn = new TucsonContextSynch(Tucson.getContext(this.getId()));
            cn.out(tid, new LogicTuple("p", new Value("hello world")));
            LogicTuple t = cn.in(tid, new LogicTuple("p",new Var("X")));
            System.out.println(t);
        } catch (Exception ex){}
    }
}

public class Test {

    public static void main(String[] args) throws Exception {
        AgentId aid = new AgentId("alice");
        new MyAgent(aid).spawn();
    }
}

```

Table 4.4: An example of `alice.tucson.api.Tucson2PLibrary` usage, providing the agent identifier as part of the description of the agent coordination context

```
:- load_library('alice.tucson.api.Tucson2PLibrary').
:- solve(go).

go:-
    enter_context([agent_id(hank)]),
    test_tc ? out(p('hello world')),
    test_tc ? in(p(X)),
    exit_context,    write(X), nl.
```

```
enter_context([agent_id(hank)])
```

an agent called `hank` requests a context to play inside a TuCSoN organisation.

To exit the context, the built-in `exit_context/0` can be used. Then, all the classical TuCSoN operations become available as built-in predicates of the form:

$$\begin{aligned}
 &TCName @ TCAAddress ? op(Tuple) \\
 &TCName ? op(Tuple) \\
 &op(Tuple)
 \end{aligned}$$

where *op* is one of `in`, `out`, `rd`, `inp`, `rdp`, `set_spec`, `get_spec`. It is worth noting that in the case the action specified by the predicate is not allowed according to current agent coordination context, the predicate fails.

In order to launch a tuProlog agent using a file containing the example Table 4.4 as logic theory, the following command it is needed to be executed:

```
java -cp tucson.jar alice.tuprolog.Agent test.pl
```

The string `hello world` will be printed on the standard output – obviously, provided that no other agent is concurrently accessing the same tuple centre, fetching the same `p(X)` tuple.

Part II

TuCSon Advanced

Chapter 5

Programming Tuple Centre Behaviour

Tuple centres are programmable tuple spaces: their behaviour can be programmed using a logic based language called ReSpecT, defining the coordination laws ruling agent interaction.

5.1 The ReSpecT Language

A ReSpecT program takes the form of a set of *reactions*:

```
reaction(Event, ( Body )).
```

Event is an extended communication event, including the execution of basic coordination primitives (`out`, `in`, `rd`, `inp`, `rdp`), and the successful execution of some ReSpecT primitives (`out_r`, `in_r`, `rd_r`, `no_r`) which will be described in the following.

Body is a sequence of ReSpecT predicates. The main predicates are listed in the Table 5.2Table 5.3: basically, they make it possible to inspect and change the content of the tuple set, by inserting / retrieving / reading associatively tuples. Also some basic Prolog predicate / builtin functors are available, mainly for making easier the manipulation of symbols and numbers, and their comparison. The full syntax of ReSpecT is described in Table 5.1, while the informal semantics of the main predicates is described in Table 5.2.

Whenever an event occurs inside a tuple centre – for instance caused by the invocation of a coordination primitive by an agent – the set of reactions triggered by the event is collected and then the body of each reaction is executed. Reactions are executed one by one, sequentially. The order of execution is non-deterministic. Actually reactions on `in` and `rd` events are triggered two times: the first time when the coordination primitive is issued by the agent (called *pre* stage), and when eventually a result (a tuple) is sent back to the agent making the request (called *post* stage). For details see the paper [11].

A simple example of reaction is:

```
reaction(out(p(X)),(
    out_r(backup(p(X))) )).
```

In this example, whenever a tuple matching the template `p(X)` is inserted in the tuple centre, another tuple `backup(p(X))` is created.

Another example is:

```
reaction(out(X),(
    in_r(n_tuples(N)),
    N1 is N + 1,
    out_r(n_tuples(N1))))).
```

```

Spec ::= Reaction*
Reaction ::= reaction( Event, ( Body ) ).
Event ::= CommunicationEvent | InternalEvent
CommunicationEvent ::=
    out(T) | in(TT) | rd(TT) | rdp(TT) | inp(TT)
InternalEvent ::=
    out_r(T) | in_r(TT) | rd_r(TT)
Body ::= Predicate{, Body}
Predicate ::= BasicPredicate | ExtendedPredicate
BasicPredicate ::=
    out_r(T) | in_r(T) | rd_r(TT) | no_r(TT) |
    pre | post |
    current_tuple(TT) | current_agent(TT) | current_op(TT) |
    current_tc(TT)
ExtendedPredicate ::=
    X is Expression |
    BooleanExpression |
    out_tc(TC, T) |
    spawn(AgentId, AgentType, [ { ArgList } ] ) |
    current_time(TT)
Expression ::= ArithmeticExpression
BooleanExpression ::= ArithmeticComparison
AgentType ::= java(AgentClassName) | prolog(TheoryFileName)
ArgList ::= T { , ArgList }

```

Table 5.1: ReSpecT Syntax

```

reaction(in(X),(
    in_r(n_tuples(N)),
    N1 is N - 1,
    out_r(n_tuples(N1))))).

reaction(inp(X),(
    in_r(n_tuples(N)),
    N1 is N - 1,
    out_r(n_tuples(N1))))).

```

Here you want to keep track of the number of tuples inside the tuple centre, by means of the tuple `n_tuples(N)`. So whenever a tuple is inserted or removed, this tuple is updated (by removing and inserting a new tuple).

The execution of a predicate can be successful or can fail: an example of a failure is when trying to retrieve a tuple by means of an `in_r` and no tuples matching the request are found. In this case the reaction is aborted *atomically*, i.e. every change apported to the tuple set by the invocation of predicate belonging to the body of the reaction is undone; in other words, the content of the tuple set is reset to the state it had before reaction execution.

Reactions can be triggered also by the successful execution of some ReSpecT predicates: `out_r`, `in_r`, `rd_r`. When the predicates is executed with success, the list of triggered reactions is collected and added to the set of the reactions whose execution is pending. If a reaction fails, also all the reactions triggered by the successful execution of the predicates of the reaction are cancelled.

5.2 Coordination as computations on the interaction space

ReSpecT is a Turing equivalent language, i.e. *any* possible computable function / algorithm can be coded in terms of a ReSpecT specification (see [3] for a demonstration). The tuple set is the data structure acting as input / output of ReSpecT computations: the fact that the language is Turing-equivalent means that given any couple of tuple set configurations C and C' it is possible to write a ReSpecT program transforming C in C' . Interactions enabled by a tuple centre

The fact that this is sufficient for creating any coordination algorithms manipulating interactions can be understood then by realising that any interactions enabled by the tuple centre can be *reified* as tuples in the set, which can be manipulated by ReSpecT specification. As an example, consider the following code:

```

reaction(in(p(X)),(
    pre,
    no_r(p(X)),
    out_r(tuple_request(p(X)))  )).

```

In this case we reify the request of a tuple (matching the template `p(X)`) currently not present in the tuple set by inserting a new tuple `tuple_requested(p(X))`.

So, the tuple set then can be adopted as a space where to build and manipulate data structures, described in terms of structured tuples. Reactions defines the algorithmic behaviour working on the data structures.

5.2.1 Basic computation constructs in ReSpecT: Selection, Iteration and Recursion

The basic constructs which typically are adopted in developing algorithms can be realised in ReSpecT by suitably composing reactions. Here we consider some basic ideas and idioms to realise selections, iterations and recursion.

Selections

Selection construct can be realised by a set of reactions with the same triggering event and containing in the first part of their body predicates succeeding on exclusive conditions, making it possible to have only one succeeding reaction at a time.

As a first simple example, suppose you want to do some operations A reacting to an event E, then to execute the operation block B or C according to the presence of a certain tuple T in the tuple set. Then, a possible reaction schema can be the following:

```
reaction( E , (
    A,
    out_r(check(T))))).

reaction(out_r(check(T)),(
    rd_r(T),
    B )).

reaction(out_r(check(T)),(
    no_r(T),
    C )).
```

As another example, suppose you want to insert a tuple `is_greater(X,Y)` whenever a tuple `compare(X,Y)` is inserted in tuple centre if X is greater than Y, supposing that X and Y are both numbers; viceversa a tuple `is_less(X,Y)` if X is less than Y. A set of reactions realising the behaviour is:

```
reaction(out(compare(X,Y)),(
    X > Y,
    out_r(is_greater(X,Y)) )).

reaction(out(compare(X,Y)),(
    X < Y,
    out_r(is_less(X,Y)) )).
```

Recursion and iterations

Recursions in ReSpecT consists in reactions triggering themselves, and it is the only means to realise iterations, suitably exploiting also selections. In the following example a factorial number is computed:

```
reaction(out(factorial(N,_)),(
    in_r(factorial(N,_)),
    out_r(fact_loop(1,N,1)))).

reaction(out_r(fact_loop(N,N,F)),(
    in_r(fact_loop(N,N,F)),
    out_r(factorial(N,F)))).

reaction(out_r(fact_loop(I,N,F)),(
    in_r(fact_loop(I,N,F)),
    N > I,
    I1 is I + 1,
    F1 is F * I1,
    out_r(fact_loop(I1,N,F1)))).
```

The strategy consists in transforming a tuple `fact_loop(I,N,F)` until a point in which the first two arguments coincide: the first is the classic index ranging all the integer numbers included within 1 and N. When the first two arguments coincide, then the third – which is transformed along with the index – contains the factorial number of the second argument.

The third reaction is recursive: it is triggered by an event (the insertion of a `fact_loop`) which the reaction generates itself.

5.3 Common mistakes

5.3.1 Binding variables more than once

ReSpecT is a logic language, following Prolog rules for what concern variable management. In particular variables can be bound to a term only once: trying to bind variables multiple times causes the predicate to fail. For instance:

```
reaction( out(T),(
    in_r(tuple_count(C)),
    C is C + 1,
    out_r(tuple_count(C)))
```

The reaction is supposed to update a tuple `tuple_count` keeping track of the number of tuples. The reaction fails since `C` is used twice. A correct version is:

```
reaction( out(T),(
    in_r(tuple_count(C)),
    C1 is C + 1,
    out_r(tuple_count(C1)))
```

< *Work in progress* >

<code>out_r(T)</code>	always succeed and as effect the logic tuple <code>T</code> is inserted in the tuple set
<code>in_r(TT)</code>	succeed if a logic tuple matching the template <code>TT</code> is present in the tuple set, and as effect the tuple is removed and unified with <code>TT</code> . Otherwise the predicate fails.
<code>rd_r(TT)</code>	succeed if a logic tuple matching the template <code>TT</code> is present in the tuple set, and as effect the tuple is unified with <code>TT</code> . Otherwise the predicate fails.
<code>no_r(TT)</code>	succeed if no logic tuple matching the template <code>TT</code> is present in the tuple set, otherwise the predicate fails.
<code>pre</code>	succeed if the event triggering the reaction is a request (<code>in</code> or <code>rd</code>) in the <i>pre</i> stage, i.e. just requested by the agent.
<code>post</code>	succeed if the event triggering the reaction is a request (<code>in</code> or <code>rd</code>) in the <i>post</i> stage, i.e. providing the result to the requesting agent.
<code>success</code>	succeed if the event triggering the reaction is a successful <code>inp</code> or <code>rdp</code>
<code>failure</code>	succeed if the event triggering the reaction is non successful <code>inp</code> or <code>rdp</code>
<code>current_agent(TT)</code>	succeed if <code>TT</code> unifies with a logic tuple denoting the identity of the agent responsible of the communication event which generated the reaction

Table 5.2: ReSpecT primitives semantics

<code>current_tuple(TT)</code>	succeed if <code>TT</code> unifies with current logic tuple specified in the event triggering the reaction
<code>current_tc(TT)</code>	succeed if <code>TT</code> unifies with a logic tuple denoting the name of current tuple centre
<code>current_time(TT)</code>	succeed if <code>TT</code> unifies with a logic tuple denoting current tuple centre virtual machine time
<code>out_tc(TC,T)</code>	succeed if <code>TC</code> is a valid and reachable tuple centre name (which can reside on a different node). The primitive has the same effect (and also is modelled as) an out operation on the specified tuple centre: the tuple <code>T</code> is inserted in the tuple centre <code>TC</code> ,
<code>spawn(AgentID,AgentType,ArgList)</code>	succeed if <code>AgentID</code> is a valid agent identifier, <code>AgentType</code> denotes a valid agent types and <code>ArgList</code> is a list of tuples. The primitive has the effect of spawning a new agent with the specified characteristics, passing some booting arguments. Currently, <code>AgentType</code> can be <code>java(ClassName)</code> or <code>prolog(FileName)</code> : in the former case, the agent is a Java agent, as an instance of the specified class (which must be found in TuCSon class path). In the latter case, the agent is a Prolog agent, whose theory is described in the specified file (which must be present in the file systems).

Table 5.3: ReSpecT primitives semantics (cont.)

Part III

Engineering systems with TuCSoN

Chapter 6

How to engineer systems: a Task-oriented approach

From a methodological point of view, engineering systems on top of TuCSoN involves basically the design and development of – on the one side – the computational parts (agents) and – on the other side – the coordination part (tuple centres protocols and behaviour). Given the uncoupling property of the coordination model, the two parts can be designed and developed almost independently. TuCSoN is an answer in particular for the coordination part. For the purpose, SODA is a methodology that can be adopted as a guide for the analysis and the design of the system [9].

6.1 Task-oriented engineering

Thinking in terms of *tasks* to be pursued, instead of control flows to be achieved, provides the appropriate abstraction level to capture and enforce the required separation of control among system components: in this context, tasks are the higher-level abstractions that drive the development of control flows, intended as lower-level mechanisms. So, identifying the relevant tasks for a given goal intrinsically implies separating control between the components charged of such tasks, since the control logic for achieving a task is embedded within a single entity. As a result, each agent interacts with the others only as much as its task needs to interact with the other tasks, thus also inherently decoupling software designers from each other.

Since reducing the agents' need to interact with each other means reducing the mutual dependencies among tasks (which is precisely what needs to be coordinated), this approach reduces the need to coordinate control among components. So, a software engineering methodology providing encapsulation of control, and identifying independent autonomous components, promotes precisely the concurrent application of the different engineering stages required by CSE.

From the viewpoint of teamwork organisation, the task metaphor allows a software team to design an interactive system by first identifying the required tasks, and then introducing as many components as needed to carry out such tasks. Interestingly, this is well adequate to the cases where the same person is in charge of designing and implementing several components: in fact, thinking of tasks instead of control flows (such as which object methods to call, which signals to send, etc.) provides precisely the design flexibility that is fundamental for an effective project management. The task metaphor granularity is also adequate to support a quick re-arrangement of the development process if the team composition changes over time: a new member can be given one or more (extra) tasks, while the tasks of a leaving member can be re-assigned to the other people in the team in much a simpler way than sharing and understanding other people's code libraries, objects, etc. Again, this situation promotes the CSE approach, since the same task can be subject of the concurrent and coordinated work of multiple designers, developers and testers.

However, defining and assigning the single tasks to components is only a part of the job: the

key issue is how to *design the “glue”* so that they actually behave as an *ensemble*. This is where engineering issues come out. If tasks and components were the only relevant entities, the system would be nothing more than the mere sum of its parts – a multitude of individuals – and interaction, merely rhyming communication, would be just the result of the random interleaving of the observable component behaviour.

But in complex systems, some tasks inevitably need to access distributed and shared resources, involving several components: that is, they naturally endorse a *social* nature. Such *social tasks* cannot, by definition, be assigned to the responsibility of an individual component: instead, their accomplishment requires an approach to system analysis, design, and development that recognises interaction as a fundamental dimension, to be accounted for since the earliest design phases. Thus, suitably engineering interaction becomes a crucial ability to be able to engineer the collective system behaviour. This is perhaps why the design of complex, interactive applications has long been considered out-of-reach for small design teams, while this is only the superficial effect of neglecting the role of interaction – governing the social tasks and rules.

6.2 Design and development of coordination in TuCSoN

TuCSoN is focused on the design, development and runtime management of the social tasks, i.e. the coordination part of the system. The design of this parts involves essentially:

- *Tuple centre cardinality and location* – establishing how many and what tuple centres to use, and their location in terms of nodes. Different strategies can be adopted, ranging from adopting a single tuple centres for enabling and governing any agent interaction, to adopting a tuple centre for each agent of the systems, acting as its private message box. Typically, you need a tuple centre for each coordination task and adopt distinct tuple centres for supporting independent coordination tasks. As general rules, (i) you can introduce *a* tuple centre whenever there are dependencies (interactions) among agents you want to capture and manage; in other words: (ii) you can introduce *a* tuple centre whenever you need a *coordination artifact* to support a set of agents in achieving a shared goal, to support their cooperative work;
- *Tuple-based protocols* – establishing the format of the tuples used and exchanged, and the related protocols, specifying the sequence of insertion / removal / reading of the tuples to/from the tuple centres;
- *Tuple centre coordinating behaviour* – programming the behaviour of the tuple centre(s) in order to realise the coordination policies / laws which characterise the social tasks.

The programming of the tuple centres can be avoided, defining all the coordination in terms of tuple based protocols (as it would happen in tuple space coordination models). The need of exploiting tuple centre programming emerge soon however when engineering complex systems, in order to get engineering properties concerning encapsulation, modularity, robustness with respect to open systems, dynamism support.

For a more detailed explanation of the steps to be adopted when engineering a systems with a coordination infrastructure you can refer to the SODA methodology [9], and as examples some case studies that have been developed in literature [15, 5].

Chapter 7

Examples and Patterns

In this chapter we present a set of simple examples and patterns, where TuCSoN is adopted as a mean to realise agent communication and coordination. The examples can be considered samples for wider classes of applications. Any example is given in its simplest form, for easing the understanding of the problem and of the solution: indeed, the examples can be the starting point for developing more robust version for real world applications.

7.1 Some Communication Patterns

7.1.1 Blackboard-based Communication

Obviously, blackboard based communication is supported naturally in TuCSoN. Agents communicate (and coordinate) by writing information on a blackboard – i.e. inserting tuples in a tuple centre –, and reading / cleaning such information – i.e. reading / retrieving tuples from the tuple centre, exploiting pattern matching. This form of communication – which is the case of tuple centre is called *generative communication* – features a number of interesting properties:

- *space uncoupling* – communication is possible even if the participants don't know where they actually are;
- *time uncoupling* – communication is possible even if the participants are not simultaneously in the same interaction context, or even alive at the same time;
- *identity uncoupling* – communication is possible even if the participants don't know who they actually are.

These properties make this approach suitable in particular for open and dynamic systems, with a dynamic set of participants, whose identity can be unknown during the communication activities.

7.1.2 Message-oriented Communication

TuCSoN can be easily adopted for realising classical message-based communication among agents. In the example a simple design is adopted (which can be easily extended, generalised according to need):

- A tuple centre `msgbox` is adopted as a shared message box or blackboard, where messages are placed and retrieved;
- A tuple `msg(Dest,Msg)` is used to represent a message: *Dest* represents the (logic/virtual) identifier (or name, nickname) of the agent receiver of the message; *Msg* represents the message;

- The protocol:

`out(msg(Dest,Msg))`, for sending a message *Msg* to an agent *Dest*;

`in(msg(Dest,?Msg))`, for receiving a message (issued by the agent *Dest* target of the message)¹;

`inp(msg(Dest,?Msg))`, for receiving a message, without blocking in the case that no message is available;

A simple sender agent, sending a string message (specified as third parameter) using a tuple centre as a message box (specified as first parameter) to an agent receiver (whose identifier is specified as first parameter):

```
import alice.tucson.api.*;
import alice.logictuple.*;

public class TestSend {
    public static void main(String[] args) throws Exception {
        // tuple centre used as a message box
        TupleCentreId tid = new TupleCentreId(args[0]);

        // receiver of the message
        Value dest = new Value(args[1]);

        // content of the message: in this case a simple string
        Value msg = new Value(args[2]);

        TucsonContextSynch cnt = new TucsonContextSynch(Tucson.getContext(new AgentId('pippo')));
        cnt.out(tid,new LogicTuple("msg",dest,msg));
    }
}
```

Note that in this case an agent is realised as a simple main, with no other classes (such as Threads or classes extending the `alice.tucson.Agent` class) involved. An example of agent execution:

```
java -cp tucson.jar TestSender msgbox@'137.204.107.188'
        cesare 'hello, see you at 5.00 pm.'
```

In this case a tuple centre called `msgbox` located on the node 137.204.107.188 (a node running the TuCSoN infrastructure) is used, `cesare` is the identifier of the agent receiver and the simple string message is 'hello, see you at 5.00 pm.'.

Then, a simple receiver agent, receiving string messages using a tuple centre as message box (specified as first parameter) specifying his identity (as second parameter) follows:

```
import alice.tucson.api.*;
import alice.logictuple.*;

import alice.tucson.api.*;
import alice.logictuple.*;

public class TestReceive {
    public static void main(String[] args) throws Exception {

        // tuple centre used as a message box
```

¹Question mark before the parameter means that it is an input parameter, so typically a variable

```

    TupleCentreId tid = new TupleCentreId(args[0]);

    // Identifier of the message reader
    Value myid = new Value(args[1]);

    TucsonContextSynch cnt = new TucsonContextSynch(Tucson.getContext(new AgentId(myid.toString())));
    LogicTuple msg = cnt.in(tid,new LogicTuple("msg",myid,new Var("Msg")));
    System.out.println("New message for "+myid+": "+msg.getArg(1));
}
}

```

An example of agent execution:

```
java -cp tucson.jar TestReceiver msgbox@'127.204.107.188' cesare
```

In this case the agent waits for messages sent to **cesare** on the **mailbox** tuple centres. Note that if no message are available, the agent waits (**in** primitive).

As a slightly different version, the following agent plays the role of a receiver, but without blocking:

```

import alice.tucson.api.*;
import alice.logictuple.*;

import alice.tucson.api.*;
import alice.logictuple.*;

public class TestReceiveNB {
    public static void main(String[] args) throws Exception {

        // tuple centre used as a message box
        TupleCentreId tid = new TupleCentreId(args[0]);

        // Identifier of the message reader
        Value myid = new Value(args[1]);

        TucsonContextSynch cnt = new TucsonContextSynch(Tucson.getContext(new AgentId(myid.toString())));

        LogicTuple msg = cnt.inp(tid,new LogicTuple("msg",myid,new Var("Msg")));
        if (msg==null){
            System.out.println("No new message for "+myid);
        } else {
            System.out.println("New message for "+myid+": "+msg.getArg(1));
        }
    }
}

```

Again for executing the agent:

```
java -cp tucson.jar TestReceiverNB msgbox@'127.204.107.188' cesare
```

Any tuple centre (on any node of the infrastructure) could be used for the example. for instance if you want to test the system locally, start a **TuCSon** node on the local machine then – for instance – use only **msgbox** as identifier for the tuple centre (localhost is used by default).

Note that in this example we exchange simple strings as messages: actually it is possible to exchange any kind of complex or structured information (including images, serialized Java objects, etc. encoded as blobs of bytes, encapsulated in strings), coded as logic tuple.

7.1.3 Remote Procedure Calls and Services

Client-server communication – as found in Remote Procedure Call or service-oriented architectures (such as Web Services) – can be modelled with a simple tuple based protocol. As a simple example consider the following design:

- A tuple centre is adopted as a mediator between clients and service providers;
- A tuple `request(Who,What)` is used to represent a service request, issued by the client *Who*, described by *What*. A tuple `result(Who,What,Result)` is used to represent a service result, issued by a service provider answering a previous request issued by the client *Who*, described by *What*.
- The protocol:

clients requests a service by inserting a tuple `request` in the tuple centre, with a `out(request(ClientID,What))`; then, they collect results by retrieving the related `request` tuple, with an `in(result(ClientID,What,?Result))`;

service providers accept a request by removing a `request` tuple with an `in(request(?ClientID,?What))` and provide results by inserting the corresponding tuple `out(result(ClientID,What,Result))`.

Of course for real world applications more robust and articulated protocols can be adopted.

The following is an example of simple service provider agent (the service provided concerns the sum of two number: request tuples are of type `request(Who,sum(X,Y))` and result tuples `result(Who,sum(X,Y),Result)`):

```
import alice.tucson.api.*;
import alice.logictuple.*;

class ServiceProviderAgent extends Thread {
    private TupleCentreId tid;

    public ServiceProviderAgent(TupleCentreId tc){
        tid = tc;
    }

    public void run() {
        try {
            TucsonContextSynch cnt = new TucsonContextSynch(Tucson.getContext(new AgentId('SPAgentS')));
            System.out.println("ServiceProviderAgent started.");
            while (true) {
                LogicTuple req = ctx.in(tid,
                    new LogicTuple("request",new Var("Who"),
                                    new Value("sum", new Var("X"),new Var("Y"))));
                System.out.println("A new request arrived "+req);
                try {
                    TupleArgument who = req.getArg(0);
                    int x = req.getArg(1).getArg(0).intValue();
                    int y = req.getArg(1).getArg(1).intValue();
                    int res = x + y;
                    ctx.out(tid,new LogicTuple("result",who,req.getArg(1),new Value(res)));
                } catch (Exception ex){
                    ex.printStackTrace();
                    System.err.println("Error in processing the request.");
                }
            }
        }
    }
}
```

```

    }
  }
} catch (Exception ex){
    System.err.println("Problems in accessing TuCSon.");
}
}
}

public class ServiceProvider {
    public static void main(String[] args) throws Exception {
        new ServiceProviderAgent(new TupleCentreId(args[0])).start();
    }
}

```

An example of service provider agent execution:

```
java -cp tucson.jar ServiceProvider services@'127.204.107.188'
```

In this case the tuple centre `services@'127.204.107.188'` is used as clients-providers mediator.

As a simple client agent:

```

import alice.tucson.api.*;
import alice.logictuple.*;

public class ServiceRequester{
    public static void main(String[] args) throws Exception {
        TucsonContextSynch cnt = new TucsonContextSynch(Tucson.getContext(new AgentId('SPAgentC')));
        TupleCentreId tid = new TupleCentreId(args[0]);

        Value who = new Value(args[1]);
        Value v0 = new Value(Integer.parseInt(args[2]));
        Value v1 = new Value(Integer.parseInt(args[3]));

        System.out.println("Agent "+who+" requesting the sum of "+v0+" and "+v1+"...");

        Value service = new Value("sum",v0,v1);
        ctx.out(tid, new LogicTuple("request",who, service));

        LogicTuple res = ctx.in(tid,new LogicTuple("result",who,service,new Var("R")));
        int result = res.getVarValue("R").intValue();
        System.out.println("The result is "+ result);
    }
}

```

An example of client agent execution:

```
java -cp tucson.jar ServiceRequester services@'127.204.107.188' client00 13 17
```

In this case the client, identified as `client00` is requesting the sum of the numbers 13 and 17, using the tuple centre `services@'127.204.107.188'`.

7.2 Simulating New Coordination Primitives

The programmability of the tuple centre makes it possible to extend the basic coordination primitive set (in, out,...) with new primitives, without altering the basic set but realising their behaviour with suitable ReSpecT reactions.

7.2.1 rd_all

The `rd_all` primitive makes it possible to collect (without removing) *all* the tuples matching a specified tuple template.

Usage: `inp(rd_all(+TupleTemplate, -List))`

Semantics: the request fails if the primitive is not supported (the ReSpecT specification has not been installed). Otherwise, the request succeeds, by returning a list (eventually empty) of the tuples matching the template.

ReSpecT specification realising the `rd_all` behaviour:

```
reaction(inp(rd_all(T,L)),(
    pre,
    out_r(rdall_collect(T,[])))).

reaction(out_r(rdall_collect(T,L)),(
    current_tuple(rdall_collect(T1,_)),
    in_r(T1),
    in_r(rdall_collect(T,L)),
    out_r(rdall_collect(T,[T1|L])))).

reaction(out_r(rdall_collect(T,_)),(
    current_tuple(rdall_collect(T1,_)),
    no_r(T1),
    in_r(rdall_collect(T,L)),
    out_r(rd_all(T,L)),
    out_r(rdall_restore(L)))).

reaction(out_r(rdall_restore([])),(
    in_r(rdall_restore([])))).

reaction(out_r(rdall_restore([H|T])),(
    in_r(rdall_restore([H|T])),
    out_r(H),
    out_r(rdall_restore(T)))).
```

7.2.2 in_all

The `in_all` primitive makes it possible to collect (and remove) *all* the tuples matching a specified tuple template.

Usage: `inp(in_all(+TupleTemplate, -List))`

Semantics: the request fails if the primitive is not supported (the ReSpecT specification has not been installed). Otherwise, the request succeeds, by removing and returning a list (eventually empty) of the tuples matching the template.

ReSpecT specification realising the `in_all` behaviour:

```
reaction(inp(in_all(T,L)),(
    pre,
    out_r(collect(T,[])))).
```

```

reaction(out_r(collect(T,L)),(
    current_tuple(collect(T1,_)),
    in_r(T1),
    in_r(collect(T,L)),
    out_r(collect(T,[T1|L])))).

```

```

reaction(out_r(collect(T,_)),(
    current_tuple(collect(T1,_)),
    no_r(T1),
    in_r(collect(T,L)),
    out_r(in_all(T,L)))).

```

7.3 Basic Synchronisation Patterns

The basic coordination primitives offers natively a good support for realising synchronisation patterns, by suitably creating protocols by means of primitives composition. However, the programmability of tuple centres makes it possible to realise (basic as well as complex) synchronisation patterns incapsulating them in tuple centre behaviour, without the need to rely on agent protocols. In other words, tuple centres can be programmed to act as *coordination artifacts*, featuring some kind of synchronising / coordinating functionality. As examples, here we consider the basic patterns as found in literature.

7.3.1 Semaphores

In order to realise a semaphore coordination artifact, there is no need of a specific ReSpecT specification.

A simple semaphore can be modelled directly as a tuple – for instance `token`; Then, any agent can invoke the classic P operation on the semaphore, i.e. semaphore request to pass, by retrieving the tuple with an `in(token)`. The classic V operation on the semaphore, i.e. semaphore release, is realised by inserting back the tuple with an `out(token)`.

A ReSpecT specification can be adopted for making this approach more robust. For instance: suppose that we want to avoid multiple signaling on the semaphore (generating multiple `token` tuples). Then the following reactions avoid the proliferation of `token` tuples:

```

reaction(out(token),(
    in_r(token),
    no_r(token),
    out_r(token))).

```

```

reaction(out(token),(
    in_r(token),
    in_r(token))).

```

In order to realise a semaphore with N agents allowed to enter simultaneously, then it is sufficient to have N token tuples.

7.3.2 Synchronisation Barrier

A typical coordination artifact needed in concurrent systems is the *synchronisation barrier*, which basically makes it possible to synchronise the activity of multiple agents. The problem concerns N agents which need to synchronise before proceeding in their individual activities: every agent executes its task until reaching the synchronisation point, and can proceed only when all the agents involved in the coordinated activities have reached it. As requirements, the agent set can be dynamic, with agents not necessarily knowing who is involved in the coordination activity.

A simple (simplified) solution in ReSpecT consists in adopting the following as individual agent protocol²:

```
out(ready) . rd(ready_all)
```

i.e. the agent inserts a `ready` tuple when he finished his activity, then wait to see a `ready_all` tuple before proceeding. Then, a tuple centre behaving as a synchronisation barrier can be obtained with the following specification:

```
reaction(out(ready), (
    in_r(ready),
    in_r(ready_agents(N)),
    N1 is N+1,
    out_r(ready_agents(N1)) )).

reaction(out_r(ready_agents(N)), (
    rd_r(barrier_size(N)),
    in_r(ready_agents(N)),
    out_r(ready_agents(0)),
    out_r(ready_all))).
```

For hypothesis, the tuple centre must contain a tuple `barrier_size`, containing current number of agents which need to be synchronised, and a tuple `ready_agents` containing the dynamic counts of agents already synchronised (starting from 0).

7.4 Other Basic Coordination Patterns

7.4.1 Interoperability

A tuple centre can be programmed in order to enable communication among agents using different communication languages and ontology, as a *syntactic and semantic mediator*. This is important in particular in open application contexts, where agents need to interact and fruitfully communicate even if they do not know each other, in terms of languages, protocols, ontology adopted. This is a typical case in complex systems where agents are developed by different developers, belonging to different organisations, and need to participate to and interact in the same application context.

7.4.2 Resource Allocation

< *Work in progress* >

7.5 Some Coordination Idioms

Coordination idioms are frequent solution adopted exploiting the peculiar features of the coordination model, in this case ReSpecT and TuCSoN.

²A process algebra-like syntax is adopted for describing protocols, with A.B meaning sequential operation, A — B parallel operations, A + B non deterministic choice between operations

7.5.1 Tuple on demand

< *Work in progress* >

7.6 Other examples

Other examples concerning forms of synchronisations and other coordination patterns can be found in the slides companion of this documents and in the TuCSoN package as demos (dining philosophers, chat systems, tictactoe, multimedia master-worker, etc).

7.6.1 Dining Philosophers

< *Work in progress* >

7.6.2 Contract Net Protocol

< *Work in progress* >

7.6.3 Workflow Engine

< *Work in progress* >

Part IV

Extensions

Chapter 8

Tuple centre persistency

8.1 Introduction

Starting with the version 1.4.0, a basic support for tuple centre *persistency* has been provided with the infrastructure. The service makes it possible to dynamically make some specific set of tuple centre persistent: when a tuple centre is persistent, its content is preserved when the node stops, either because of a manual shutdown, or a failure/crash. When the node is started again, the tuple centres for which the persistence was on are automatically restored.

8.2 Enabling and disabling persistency

The persistency service can be exploited by suitably interacting with the tuple centre \$ORG of the node. The protocol of the service follows:

- To enable the persistency:

```
'$ORG' @ Node ? out(cmd(enable_persistency(TCName)))
```

Node is the TuCSoN node; *TCName* is a template defining the set of tuple centres to make persistent. It can be directly a tuple centre name, as well as a template with variables defining a set of names. The effect of the invocation is to make persistent every tuple centre whose name matches the template specified.

In order to check if the command has been successful:

```
'$ORG' @ Node ? in(cmd(enable_persistency(TCName)), Result)
```

If the command has been successful, *Result* is bound to the atom `ok`; otherwise, `unknown` is returned if the command was unknown, `failed` if the command was unsuccessful.

- To disable the persistency:

```
'$ORG' @ Node ? out(cmd(disable_persistency(TCName)))
```

Node is the TuCSoN node; *TCName* is a template defining the set of tuple centres (previously made persistent) to make not persistent.

As in the previous case, in order to check if the command has been successful:

```
'$ORG' @ Node ? in(cmd(disable_persistency(TCName)), Result)
```

If the command has been successful, *Result* is bound to the atom `ok`; otherwise, `unknown` is returned if the command was unknown, `failed` if the command was unsuccessful.

- To check if a tuple centre is persistent:

```
'$ORG' @ Node ? rdp(is_persistent(TCName))
```

Node is the TuCSoN node; *TCName* is the name of the tuple centre to check. The request returns successfully if the *TCName* is persistent.

8.3 Persistent strategy

The basic strategy adopted in this version creates a textual file for each tuple centre made persistent, with name equals to `tc_` plus the name of the tuple centre. The files are stored in a subdirectory called `persistent` where the TuCSoN node is launched. Each textual file contain a snapshot of the tuple centre content when the persistency has been enabled, and the list of all the operations executed on the tuple set, changing its content. When a node is installed (launched), all the textual files stored in the `persistent` directory are inspected and the related tuple centre restored.

Chapter 9

Timed ReSpecT

9.1 Introduction

In most application scenarios characterised by a high degree of openness and dynamism, coordination tasks need to be time-dependent. On the one hand, it is very useful to specify (and then enforce) given levels of liveness and of quality of service — e.g. requiring agents to interact with the coordination artifact at a minimum/maximum frequency. On the other hand, temporal properties are also fundamental aspects concerning interception of violations in the agent-artifact contract: an agent might be required to provide a service within a given deadline, or might require the artifact to do the same. Accordingly, the version 1.4.0 of ReSpecT has been extended to include the capabilities of managing time event, in order to support the definition and enactment of time-aware coordination policies. The basic idea is to exploit the programmability of the coordination medium extended with a temporal framework to get the capability of modelling *any* time-based coordination patterns, realised directly by specifying a suitable behaviour of the artifact.

9.2 Extending ReSpecT with Time

First of all, the model is extended with a notion of current time of the tuple centre Tc : each tuple centre has its own clock, which defines the passing of time¹. Actually, tuple centre time is a physical time, but its value is considered to be constant during the execution of an individual reaction: in other words, we assume that Tc refers to the time when the reaction started executing. This choice is coherent with ReSpecT philosophy concerning reactions, which are meant to be executed atomically (in the case of successful reactions).

In order to get Tc in ReSpecT programs a new primitive is introduced:

```
current_time(?Tc)2
```

This primitive (predicate) is successful if Tc (typically a variable) unifies with the current tuple centre time Tc . As an example, the reaction specification tuple

```
reaction(in(p(X)),(  
    current_time(Tc),  
    out_r(request_log(Tc,p(X)))  
)).
```

inserts a new tuple with timing information each time a request to retrieve a tuple $p(X)$ is executed, realising a temporal log of the requests.

¹In current implementation the temporal unity is the millisecond

²A Prolog notation is adopted for describing the modality of arguments: + is used for specifying input argument, - output argument, ? input/output argument, @ input argument which must be fully instantiated

The model is then extended with the notion of *trap event* or simply *trap*, which is an event generated when the tuple centre reaches a specific time point. A trap occurs because of a (*trap*) *source*, characterised by a unique identifier *ID*, a time *Te* and a description tuple *Td*. The language is extended with the possibility to generate and manipulate trap events and sources. In particular we introduce the two following features:

- internally in the tuple centre, a coordination law (i.e. one or more reaction specification tuples) might install a trap source, which causes a trap to occur at a specific time. For instance, we may want to generate a trap described by the tuple `expired(T)` a certain interval *LeaseTime* after the insertion of a tuple `leased(T)`;
- the tuple centre reacts to a trap event analogously to communication events, by means of proper reaction specification tuples. In the case above, we may want the tuple *T* to be removed when the trap described by `expired(T)` occurs.

In order to support trap generator installation, the language is extended with two new primitives:

```
new_trap(-ID,@Te,+Td)

kill_trap(@ID)
```

The first is successful if *Te* is an integer equal or greater than zero. Its effect is to install a new trap source — with *ID* as identifier — which enters a queue of installed sources. When tuple centre time *Tc* time will be equal or greater than current time plus *Te*, a trap event described by the tuple *Td* will be then generated and inserted into the queue of triggered trap events, whereas its source is deinstalled — i.e. removed from its queue. Notice that because of the success/failure semantics of ReSpecT semantics, if the reaction including an invocation to primitive `new_trap` fails, no trap source is installed, actually. An example involving the `new_trap` primitive is as follows:

```
reaction(out(leased(T,LeaseTime)),(
  new_trap(_,LeaseTime,expired(T))
)).
```

The reaction is triggered when a tuple matching `leased(T,LeaseTime)` is inserted, and it installs a new trap source which will generate a trap described by the tuple `expired(T)` after *LeaseTime* units from then. Primitive `kill_trap` is instead used to deinstall a source given its identifier: such a primitive fails if not installed sources has is characterised by the identifier provided.

Then, the language has been extended with the possibility to write reactions triggered by the occurrence of trap events. The syntactical and semantic models of trap reactions are analogous to the reactions to communication events:

```
reaction( trap(Tuple), Body)
```

Body specifies the set of actions to be executed when a trap with a description tuple matching the template *Tuple* occurs. In the following simple example

```
reaction(trap(expired(T)),( in_r(T) )).
```

when a trap described by a tuple matching the template `expired(T)` occurs, the tuple specified in *T* is removed from the tuple set. Notice that if the tuple is not present the `in_r` fails causing the whole reaction to fail — as the trap event is occurred, however, the trap source is erased.

Trap events are listened one by one as soon as the tuple centre is not executing a reaction; that is — according to the tuple centre semantics [12, 11] — when it is in the idle state, or between a listening and a speaking stage, or during a reacting stage (between the execution of two reactions). When a trap event is listened, it is first removed from the trap event queue, the set of the reactions it triggers is determined — by matching the reaction head with the trap description tuple — and then executing

sequentially all such reactions. As for the ReSpecT reacting stage, the order of execution of the reactions is not deterministic.

An important semantic aspect of this extension concerns the priority of reactions fired by external communication events (standard execution) with respect to those of trap events (trap execution). The model and implementation described here feature higher priority of reactions fired by trap events. This means that if during the standard executions of a reaction chain a trap event occurs, the chain is broken, and the reactions fired by the trap are executed. It's worth noting that the individual reactions are still atomic, not interruptible as in the basic ReSpecT model: traps event in the trap queue are listened (and related reactions executed) after the completion of any reaction eventually in execution. Then, chains of reactions can be broken, not individual reactions. This is fundamental in order to preserve the semantic properties of ReSpecT model [11]. Also reactions triggered by a trap event are atomic, and they cannot be interrupted or suspended: in other words, trap handlers are not interruptible and cannot be nested.

The possibility of breaking reaction chains is important to build robust coordinating behaviour, in particular with respect to possible bugs generating terminating reaction chains.

Nevertheless, it is worth mentioning here that other semantics are possible and interesting. By giving higher priority to the standard execution, one ensures that traps never interfere with it. In exchange of the better isolation of code achieved, in this case one can no longer guarantee the same timing constraints: trap executions must wait for the standard execution to complete. Notice that such aspects are mostly orthogonal to the actual applicability of temporal coordination laws as shown e.g. in next section. Moreover, a straightforward generalisation of our model can be realised by specifying the priority level of a trap (higher, lower, or equal to the that of external communication events) at the time its source is installed³.

9.3 Examples

In this section we describe some simple examples of how temporal coordination primitives and coordination laws can be modelled on top of extended ReSpecT. It's worth noting that these examples – even if simple – appear in several research work in literature as a core of timing features extending the basic model; typically, in the literature there is a specific extension for each timing feature described here: on the contrary, we remark the generality of our approach, which is meant to support these and several other time-based coordination patterns on top of the same model.

9.3.1 Timed Requests

In this first example we model a timed `in` primitive, i.e. an `in` request that keeps blocked only for a maximum amount of time. An agent issues a timed `in` by executing primitive `in(timed(@Time, ?Template, -Res))`. If a tuple matching *Template* is inserted within *Time* units of time, the requested tuple is removed and taken by the agent as usual with *Res* being bound to the `yes` atom. Conversely, if no matching tuples are inserted within the specified time, *Res* is bound to `no` atom. Table 9.1 reports the ReSpecT specification which makes it possible to realise the behaviour of this new primitive. When the `in` request is issued, if a tuple matching the template is present a proper tuple satisfying the request is created (reaction 1). Instead, if no tuple is found, a trap source is installed for generating a trap at the due time (reaction 2). Also, a tuple `trap.info` is inserted in the tuple set, reifying information about the installed trap source, required for its possible removal. If a tuple matching a template of a pending timed `in` is inserted on time, the related trap source is removed and a proper tuple matching the timed `in` request is inserted (reaction 4). Finally, if the trap occurs – meaning that no tuples have been inserted on time matching a pending timed `in` – then a tuple matching the timed `in` request carrying negative result is inserted in the tuple set (reaction 3).

³This interesting feature which is subject of current research is not described in this paper for brevity.

```

1 reaction( in(timed(Time,Tuple,Res)), (
    pre, in_r(Tuple),
    out_r(timed(Time,Tuple,yes))))).

2 reaction( in(timed(Time,Tuple,Res)), (
    pre,no_r(Tuple),
    new_trap(ID,Time,expired_in(Time,Tuple)),
    out_r(trap_info(ID,Time,Tuple)) ).

3 reaction( trap(expired_in(Time,Tuple)),(
    in_r(trap_info(ID,Time,Tuple)),
    out_r(timed(Time,Tuple,no)) ).

4 reaction( out(Tuple),(
    in_r(trap_info(ID,Time,Tuple)),
    kill_trap(ID),
    out_r(timed(Time,Tuple,yes)) ).

```

Table 9.1: ReSpecT specification for modelling a timed in primitive

```

1 reaction( out(leased(Time,Tuple)), (
    new_trap(ID,Time,lease_expired(Time,Tuple)),
    in_r(leased(Time,Tuple)),
    out_r(outl(ID,Time,Tuple)) ).

2 reaction( rd(Tuple),( pre,
    rd_r(outl(ID,.,Tuple)),
    out_r(Tuple) ).

3 reaction( rd(Tuple),(post,
    rd_r(outl(ID,.,Tuple)),
    in_r(Tuple) ).

4 reaction( in(Tuple),( pre,
    in_r(outl(ID,.,Tuple)),
    out_r(Tuple),
    kill_trap(ID) ).

5 reaction( trap(lease_expired(Time,Tuple)), (
    in_r(outl(ID,Time,Tuple))))).

```

Table 9.2: ReSpecT specification for modelling tuples with a lease time

```

1  reaction(in(all_timed(Time,Tuple,OutList)),(
    new_trap(ID,Time,inat(Time,Tuple,OutList)),
    out_r(current_in.all(ID,Time,Tuple,[])),
    out_r(remove_in.all(ID))))).
2  reaction( out_r(remove_in.all(ID)),(
    in_r(remove_in.all(ID)),
    rd_r(current_in.all(ID,Time,Tuple,L)),
    in_r(Tuple),
    in_r(current_in.all(ID,Time,Tuple2,L)),
    out_r(current_in.all(ID,Time,Tuple2,[Tuple|L])),
    out_r(remove_in.all(ID)))).
3  reaction( out_r(remove_in.all(ID)),(
    in_r(remove_in.all(ID)),
    rd_r(current_in.all(ID,_,Tuple,_)),
    no_r(Tuple))).
4  reaction( out(Tuple),(
    in_r(current_in.all(ID,_,Tuple,L)),
    in_r(Tuple),
    out_r(current_in.all(ID,_,Tuple,[Tuple|L])))).
5  reaction( trap(inat(Time,Tuple,OutList)), (
    in_r(current_in.all(ID,Time,Tuple,L)),
    out_r(all_timed(Time,Tuple,L)))).

```

Table 9.3: ReSpecT specification mimicking an `inall` with a duration time

9.3.2 Tuples in Leasing

In this example we model the notion of *lease*, analogously to the lease notion in models such as JavaSpaces [6] and TSpaces [17]. Tuples can be inserted in the tuple set specifying a lease time, i.e. the maximum amount of time for which they can reside in the tuple centre before automatic removal.

An agent insert a tuple with a lease time by issuing an `out(leased(@Time,@Tuple))`. Table 9.2 shows the ReSpecT specification programming the tuple centre with the desired leasing behaviour. When a tuple with a lease time is inserted in the tuple centre, a trap source is installed for generating a trap when the tuple centre time reaches the lease due time (reaction 1). Also a tuple `out1` is inserted in the tuple set with the information on the trap source and the leased tuple (note that the flat tuple with the lease time is not directly present in the set). Then, for each `rd` issued with a template matching a leased tuple, a flat tuple satisfying the request is first inserted in the tuple set (reaction 2), and then removed after the `rd` has been satisfied (reaction 3). An `in` request instead causes directly the removal of the lease tuple and of the trap source (reaction 4). Finally, if a trap event occurs (meaning that the lease time of a tuple expired), the `out1` tuple carrying information about the presence of the leased tuple is removed (reaction 5).

9.3.3 Dining Philosophers with Maximum Eating Time

9.3.4 Dining Philosophers with Maximum Eating Time

The *dining philosopher* is a classical problem used for evaluating the expressiveness of coordination languages in the context of concurrent systems. In spite of its formulation, it is generally used as an archetype for non-trivial resource access policies. The solution of the problem in ReSpecT consists in using a tuple centre for encapsulating the coordination policy required to decouple agent requests from single requests of resources — specifically, to encapsulate the management of *chopsticks* (for details refer to [12]).

Each philosopher agent (*i*) gets the two needed chopsticks by retrieving a tuple `chops(C1,C2)`, (*ii*) eats for a certain amount of time, (*iii*) then he provides back the chopsticks by inserting the tuple `chops(C1,C2)` in the tuple centre, and (*iv*) finally he starts thinking until next dining cycle. A pseudo-code reflecting this interactive behaviour is the following:

```

% a request of the chopsticks is reified with a
% required tuple
1  reaction(in(chops(C1,C2)),(pre,out_r(required(C1,C2)))).

% if both the chopsticks are available, a chops
% tuple is generated
2  reaction(out_r(required(C1,C2)),(
    in_r(chop(C1)),in_r(chop(C2)),out_r(chops(C1,C2)))).

% with the retrieval of the chops tuple,
% the chopsticks request is removed
3  reaction(in(chops(C1,C2)),
    (post,in_r(required(C1,C2)))).

% the release of a chops tuple still valid (on time)
% causes the insertion of individual chopsticks,
% represented by the two chop tuples
4  reaction(out(chops(C1,C2)), (
    current_agent(AgentId),
    no_r(invalid_chops(AgentId,C1,C2)),
    in_r(chops(C1,C2)),out_r(chop(C1)),out_r(chop(C2)))).

% a chops tuple is generated if there is
% a pending request, and both chop tuples
% are actually available
5  reaction(out_r(chop(C1)), (rd_r(required(C1,C)),
    in_r(chop(C1)),in_r(chop(C)),out_r(chops(C1,C)))).
6  reaction(out_r(chop(C2)), (rd_r(required(C,C2)),
    in_r(chop(C)),in_r(chop(C2)),out_r(chops(C,C2)))).

% a chopsticks request causes also creating a
% new trap generator, keeping track of its
% information % in the chops_pending_trap tuple
7  reaction(in(chops(C1,C2)),( pre,
    rd_r(max_eating_time(Tmax)),
    new_trap(ID,Tmax, expired(C1,C2)),
    current_agent(AgentId),
    out_r(chops_pending_trap(ID,AgentId,C1,C2)))).

% when chops are released on time, the trap
% generator is removed
8  reaction(out(chops(C1,C2)),(
    in_r(chops_pending_trap(ID,C1,C2)),
    kill_trap(ID))).

% trap generation causes the insertion back
% of the missing tuples and the insertion of tuple
% keeping track of the invalid chops
9  reaction(trap(expired(C1,C2)),(
    no_r(chop(C1)), no_r(chop(C2)),
    current_agent(AgentId),
    in_r(chops_pending_trap(ID,AgentId,C1,C2)),
    out_r(invalid_chops(AgentId,C1,C2)),
    out_r(chop(C1)), out_r(chop(C2)))).

% chopsticks released that are invalid (due to
% time expiration) are immediately removed
10 reaction(out(chops(C1,C2)), (
    current_agent(AgentId),
    in_r(invalid_chops(AgentId,C1,C2)),
    in_r(chops(C1,C2)))).

```

Table 9.4: ReSpecT specification for coordinating dining philosophers with a maximum eating time

```

while (true){
  think();
  in(chops(C1,C2));
  eat();
  out(chops(C1,C2));
}

```

The coordination specification in ReSpecT (first 6 reactions of Table 9.4, bottom) mediates the representation of the resources (`chops` vs. `chop` tuples), and most importantly avoid deadlocks among the agents.

Here we extend the basic problem by adding a further constraint: the maximum time which philosophers can take to eat (i.e. to use the resources) is given, stored in a tuple `max_eating_time(MaxEatingTime)` in the tuple centre. To keep the example simple, if this time is exceeded, the chopsticks are regenerated in the tuple centre, avoiding the starvation of the philosophers waiting for them, and the chopsticks eventually inserted out of time are removed.

The solution to this problem using the extended ReSpecT model accounts for adding only the ReSpecT specification (the agent code and related protocols are untouched) with the reactions 7–10 described in Table 9.4 (bottom), and extending reaction 4 with the part in *italics*. Essentially, the new reactions install a new trap source as soon as a philosopher retrieves his chopsticks (reaction 7). If the philosopher provides the chopsticks back in time (before the occurrence of the trap), then the trap source is removed (reaction 8). Otherwise, if the trap event occurs, the triggered trap reaction recreates the missing chopsticks tuples in the tuple centre and inserts a tuple `invalid_chops` which prevent chopsticks insertion out of time (reaction 9). This prevention is realised by checking the existence of the tuple `invalid_chops` when the tuple `chops` are released by a philosopher (reaction 10).

It is worth noting that keeping track of the maximum eating time as a tuple (`max_eating_time` in the example) makes it possible to easily change it dynamically, while the activity is running; this can be very useful for instance in scenarios where this time need to be adapted (at runtime) according to the workload and, more generally, environmental factors affecting the system.

Finally, it's worth remarking that the approach is not meant to alter the autonomy of the agent, for instance by means of some form of preemption in the case of timing violations; on the contrary – as a coordination model – all the constraints and (timed based) rule enforcing concerns the interaction space.

9.3.5 An Artifact for Timed Contract Net Protocols

As a final example, we describe a coordination artifact modelling and embodying the coordinating behaviour of a time-aware Contract Net Protocol (CNP). CNP is a well-known protocol in MAS, used as basic building block for building more articulated protocols and coordination strategies [16]. Following [7], we consider the CNP in a task allocation scenario: a master announces a task (service) to be executed, potential workers interested provide their bids, the announcer collects the bid and selects one; after confirming his bid, the awarded bidder becomes the contractor, taking in charge of the execution of the task and finally providing task results.

We extend the basic version with some timing constraints. In particular we suppose that: *(i)* the bidding stage has a duration, established at a “contract” level; *(ii)* there is a maximum time for the announcer for communicating the awarded bidder; *(iii)* there is a maximum time for the awarded bidder for confirming the bid and becoming the contractor; *(iv)* there is a maximum time for the contractor for executing the task.

According to our approach, a coordination artifact can be used to embody the coordinating behaviour of the time-aware CNP, fully encapsulating the social/contractual rules defining protocols steps and governing participant interaction, including temporal constraints. The coordination artifact is realised as a tuple centre – called `tasks` –, programmed with the ReSpecT specification reported in Table 9.6.

```

tasks ? out(announcement(task(TaskId,TaskInfo,MaxExecTime)))
tasks ? in(bids(TaskId,BidList))
Bid ← selectWinner(BidList)
tasks ? out(awarded_bid(TaskId,AgentId))
tasks ? in(task_done(TaskId,Result,Duration))

```

```

tasks ? rd(announcement(task(TaskId,TaskInfo,MaxExecTime)))
MyBid ← evaluate(TaskInfo)
tasks ? out(bid(TaskId,MyId,MyBid))
tasks ? in(bid_result(TaskId,MyId,Answer))
if (Answer=='awarded') {
    tasks ? out(confirm_bid(MyId))
    Result ← perform(TaskInfo)
    tasks ? out(task_result(TaskId,MyId,Result))
}

```

Table 9.5: Sketch of the behaviour of the agents participating to the timed Contract Net Protocol: masters (*Top*) and workers (*Bottom*)

Table 9.5 shows the pseudo-code representing the interactive behaviour of the master (top) and workers (bottom).

The usage protocol of the artifact for the master consists in: making the announcement (by inserting a tuple **announcement**), collecting the bids (by retrieving the tuple **bids**), selecting and informing the awarded bidder (by inserting the tuple **awarded_bid**) and, finally, collecting the result (by retrieving the tuple **task_done**); for the workers, the usage protocol accounts for reading the announcement (by reading the tuple **announcement**), evaluating the proposal and providing a bid (by inserting a tuple **bid**), reading the master decision (by retrieving the tuple **bid_result**), and – in the case of awarding – confirming the bid (by inserting the tuple **confirming_bid**), performing the task and, finally, providing the results (by inserting the tuple **task_result**).

The artifact behaviour in ReSpecT described in Table 9.6 reflects the various stages of the CNP protocol, and traps are used for modelling the timing constraints related to the various stages: from bidding, to awarding, confirming, and task execution. A brief description of the artifact behaviour follows: when a new announcement is done (reaction 1), the information about the new CNP are created (tuple **task_todo** and **cnp_state**) and a new trap source is installed, generating a trap when the bidding time is expired. At the trap generation (reaction 2) – meaning that the bidding stage is closed – all the bids inserted are collected (reaction 3), the information concerning the protocol state updated, and a new trap source is installed, generating a trap when the awarding time is expired. If the master provides information about the awarded bidder before this trap generation, the trap source is killed, the tuples concerning awarded and non-awarded bidders are generated (reactions 5, 8, 9), and a new trap source for managing confirmation expire is installed (reaction 5). If no awarded bidder is provided on time or a wrong (unknown) awarded is communicated, the tuple reporting the CNP state is updated accordingly, reporting the error (reactions 4, 6, 7). If the awarded bidder confirms on time his bid (reaction 10), the execution stage is entered, by updating the CNP state properly and installing a new trap generator for keeping track of task execution time expiration. Otherwise, if the confirm is not provided on time, the related trap event is generated and listened (reaction 11), aborting the activity and updating accordingly the CNP state tuple. Finally, if the contractor provides the task result on time (reaction 13), the trap generator for task execution is killed, the tuples concerning the terminating CNP are removed and the result information are prepared for being retrieved by the master. Otherwise, if the contractor does not provide information on time, the trap is generated and the artifact state is updated accordingly, reporting the error (reaction 12).

```

% When an announcement is made, a trap generator is
% installed for generating a timeout for bidding time
1 reaction(out(announcement(task(Id,Info,MaxTime))), (
    out_r(task_todo(Id,Info,MaxTime)),
    out_r(cnp_state(collecting_bids(Id))),
    rd_r(bidding_time(Time)),
    new_trap(.,Time,bidding_expired(Id)))).

% When the bidding time has expired, the master can
% collect the bids for choosing the winner. A trap
% generator is installed for defining the maximum
% awarding time
2 reaction(trap(bidding_expired(TaskId)), (
    in_r(announcement(.)),
    in_r(cnp_state(collecting_bids(TaskId))),
    out_r(collected_bids(TaskId,[])),
    out_r(cnp_state(awarding(TaskId))),
    rd_r(awarding_time(Time)),
    new_trap(.,Time,awarding_expired(TaskId)))).
3 reaction(out_r(collected_bids(TaskId,L)), (
    in_r(bid(TaskId,AgentId,Bid)),
    out_r(bid_evaluated(TaskId,AgentId,Bid)),
    in_r(collected_bids(TaskId,L)),
    out_r(collected_bids(TaskId,
        [bid(AgentId,Bid)|L])))).

% When the awarding time has expired, the bidders
% are
% informed of the results. If no winner has been
% selected the protocol enters in an error state,
% otherwise the protocol enters in the confirming
% stage, setting up a maximum time for it
4 reaction(trap(awarding_expired(TaskId)), (
    in_r(cnp_state(awarding(TaskId))),
    out_r(check_awarded(TaskId)))).
5 reaction(out_r(check_awarded(TaskId)), (
    in_r(check_awarded(TaskId)),
    rd_r(awarded_bid(TaskId,AgentId)),
    in_r(bid_evaluated(TaskId,AgentId,Bid)),
    out_r(result(TaskId,AgentId,awarded)),
    out_r(cnp_state(confirming_bid(TaskId,AgentId))),
    rd_r(confirming_time(Time)),
    new_trap(ID,Time,confirm_expired(TaskId)),
    out_r(confirm_timer(TaskId,ID)),
    out_r(refuse_others(TaskId)))).
6 reaction(out_r(check_awarded(TaskId)), (
    in_r(check_awarded(TaskId)),
    rd_r(awarded_bid(TaskId,AgentId)),
    no_r(bid_evaluated(AgentId,Bid)),
    out_r(cnp_state(aborted(TaskId,wrong_awarded))))).
7 reaction(out_r(check_awarded(TaskId)), (
    in_r(check_awarded(TaskId)),
    no_r(awarded_bid(TaskId,AgentId)),
    out_r(cnp_state(aborted(TaskId,award_expired))))).

8 reaction(out_r(refuse_others(TaskId)), (
    in_r(bid_evaluated(TaskId,AgentId,Bid)),
    out_r(result(TaskId,AgentId,'not-awarded')),
    out_r(refuse_others(TaskId)))).
9 reaction(out_r(refuse_others(TaskId)), (
    in_r(refuse_others(TaskId)))).

% At the arrival of the confirm from the awarded
% bidder, a timeout trap is setup for checking the
% execution time of the task
10 reaction(out(confirm_bid(TaskId,AgentId)), (
    in_r(confirm_bid(TaskId,AgentId)),
    in_r(cnp_state(confirming_bid(TaskId,AgentId))),
    current_time(StartTime),
    out_r(cnp_state(executing_task(TaskId,StartTime))),
    in_r(confirm_timer(TaskId,IdT)),
    kill_trap(IdT),
    rd_r(task_todo(TaskId,.,MaxTime)),
    new_trap(IdT2,MaxTime,execution_expired),
    out_r(execution_timer(TaskId,IdT2)))).

% The occurrence of the confirm expired trap means
% that the confirm from the awarded bidder has not
% arrived on time, causing the protocol to be aborted
11 reaction(trap(confirm_expired(TaskId)), (
    in_r(cnp_state(confirming_bid(TaskId,AgentId))),
    in_r(confirm_timer(TaskId,._)),
    rd_r(awarded_bid(TaskId,AgentId)),
    out_r(cnp_state(aborted(TaskId,
        confirm_expired(AgentId)))))).

% The occurrence of the execution expired trap means
% that the awarded bidder has not completed the
% task on time, causing the protocol to be aborted
12 reaction(trap(execution_expired(TaskId)), (
    in_r(cnp_state(executing_task(TaskId,StartTime))),
    in_r(execution_timer(TaskId,._)),
    rd_r(awarded_bid(TaskId,AgentId)),
    current_time(Now),
    Duration is Now - StartTime,
    out_r(cnp_state(aborted(TaskId,
        execution_expired(AgentId,Duration))))).

% The awarded bidder provided task result on time
% terminating correctly the protocol
13 reaction(out(task_result(TaskId,AgentId,Result)), (
    in_r(task_result(TaskId,AgentId,Result)),
    in_r(awarded_bid(TaskId,AgentId)),
    in_r(execution_timer(TaskId,Id)),
    kill_trap(Id),
    in_r(cnp_state(executing_task(TaskId,StartTime))),
    in_r(task_todo(TaskId,Info,MaxTime)),
    current_time(Now),
    Duration is Now - StartTime,
    out_r(task_done(TaskId,Result,Duration)))).

```

Table 9.6: Behaviour of the artifact realising a timed CNP, encoded in the ReSpecT language

Chapter 10

Extending TuCSoN with Organisation and Security Support: RBAC

TuCSoN model has been recently extended with a basic integrated support for organisation and security modelling, based on a *Role-Based Access Control* architecture (RBAC).

Such a support will be implemented in next TuCSoN release.

Appendix A

Related Documents and Publications

The updated list of articles, papers and documents about TuCSoN can be found at

<http://www.alice.unibo.it/projects/tucson>

- [12] – describes the tuple centre coordination model;
- [11] – describes the ReSpecT language used to specify behaviour of tuple centre as coordination laws;
- [2] – describes the fundamental concept of programmable coordination medium;
- [3] – describes the expressive power of ReSpecT tuple centre;
- [13] – introduces TuCSoN as Coordination Model/Infrastructure for Internet Application Development;
- [8] – describes the concept of hybrid coordination model;
- [14] – book about coordination models, technologies and applications for Internet agents;
- [1] – describes TuCSoN topology and security model suitable for mobile agents;
- [9] – describes the SODA software engineering methodology, whose design and development part can be supported by TuCSoN;
- [10] – introduces the notion of Agent Coordination Context;
- [4] provides a general overview about tuProlog project and technology;

Appendix B

Document History

Date	Action	Responsible
2002-04-05	The document is created	aricci
2002-31-05	minor revision	edenti
2002-09-21	major revision	aricci
2004-07-10	major revision	aricci
2004-10	major revision	aricci
2006-11	updates to 1.4.5	aricci

Appendix C

Planned Development of this Document

The following arguments/chapters are work in progress and will be added as soon as possible:

- Detailed description of how to (dynamically) configure a TuCSoN node, in terms of agent coordination contexts supported, and so security and quality of services issues;
- Descriptions of the basic ontology used for the standard part of context context description, and, more generally, of issues concerning agent coordination contexts;
- Tutorial and example about the design and development of coordination programs in ReSpecT language;
- Description of some of the demos provided with TuCSoN;
- TuCSoN and SODA [9] methodology: some examples of how to exploit the TuCSoN model with the SODA agent oriented software engineering methodology for engineering systems.

Bibliography

- [1] M. Cremonini, A. Omicini, and F. Zambonelli. Ruling agent motion in structured environments. In M. R. Bubak, H. Afsarmanesh, R. Williams, and B. Hertzberger, editors, *High Performance Computing and Networking*, volume 1823 of *LNCS*, pages 187–196. Springer-Verlag, 2000. 8th International Conference (HPCN Europe 2000), 8–10 May 2000, Amsterdam (NL), Proceedings.
- [2] E. Denti, A. Natali, and A. Omicini. Programmable coordination media. In D. Garlan and D. Le Métayer, editors, *Coordination Languages and Models – Proceedings of the 2nd International Conference (COORDINATION’97)*, volume 1282 of *LNCS*, pages 274–288, Berlin (D), 1–3 Sept. 1997. Springer-Verlag.
- [3] E. Denti, A. Natali, and A. Omicini. On the expressive power of a language for programming coordination media. In *1998 ACM Symposium on Applied Computing (SAC’98)*, pages 169–177, Atlanta (GA), 27 Feb. – 1 Mar. 1998. ACM. Track on Coordination Models, Languages and Applications.
- [4] E. Denti, A. Omicini, and A. Ricci. tuProlog: A light-weight Prolog for Internet applications and infrastructures. In I. Ramakrishnan, editor, *Practical Aspects of Declarative Languages*, volume 1990 of *LNCS*, pages 184–198. Springer-Verlag, 2001. 3rd International Symposium (PADL 2001), Las Vegas (NV), 11–12 Mar. 2001, Proceedings.
- [5] E. Denti, A. Ricci, and R. Rubino. Integrating and orchestrating services upon an agent coordination infrastructure. In A. Omicini, P. Petta, and J. Pitt, editors, *Engineering Societies in the Agents World IV*, LNCS. Springer-Verlag, 2004. 4th International Workshop (ESAW 2003), London, UK, 29–31 Oct. 2003. Revised Papers.
- [6] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces: Principles, Patterns, and Practice*. The Jini Technology Series. Addison-Wesley, 1999.
- [7] M. Huhns and L. M. Stephens. Multiagent systems and societies of agents. In G. Weiss, editor, *Multiagent Systems – A Modern Approach to Distributed Artificial Intelligence*, pages 79–118. MIT Press, 1999.
- [8] A. Omicini. Hybrid coordination models for handling information exchange among Internet agents. In A. Bonarini, M. Colombetti, and P. L. Lanzi, editors, *AI*IA 2000 Workshop “Agenti intelligenti e Internet: teorie, strumenti e applicazioni”*, pages 1–4, Milano (Italy), 13 Sept. 2000.
- [9] A. Omicini. SODA: Societies and infrastructures in the analysis and design of agent-based systems. In P. Ciancarini and M. J. Wooldridge, editors, *Agent-Oriented Software Engineering*, volume 1957 of *LNCS*, pages 185–193. Springer-Verlag, 2001. 1st International Workshop (AOSE 2000), Limerick, Ireland, 10 June 2000, Revised Papers.
- [10] A. Omicini. Towards a notion of agent coordination context. In D. Marinescu and C. Lee, editors, *Process Coordination and Ubiquitous Computing*, pages 187–200. CRC Press, 2002.

- [11] A. Omicini and E. Denti. Formal ReSpecT. In A. Dovier, M. C. Meo, and A. Omicini, editors, *Declarative Programming – Selected Papers from AGP’00*, volume 48 of *Electronic Notes in Theoretical Computer Science*, pages 179–196. Elsevier Science B. V., 2001.
- [12] A. Omicini and E. Denti. From tuple spaces to tuple centres. *Science of Computer Programming*, 40(2), July 2001.
- [13] A. Omicini and F. Zambonelli. Coordination for Internet application development. *Autonomous Agents and Multi-Agent Systems*, 2(3):251–269, Sept. 1999. Special Issue: Coordination Mechanisms for Web Agents.
- [14] A. Omicini, F. Zambonelli, M. Klusch, and R. Tolksdorf, editors. *Coordination of Internet Agents: Models, Technologies, and Applications*. Springer-Verlag, Mar. 2001.
- [15] A. Ricci, A. Omicini, and E. Denti. Virtual enterprises and workflow management as agent coordination issues. *International Journal of Cooperative Information Systems*, 11(3/4):355–379, Sept./Dec. 2002. Special Issue: Cooperative Information Agents – Best Papers of CIA 2001.
- [16] R. G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. In *Proceedings of the 1st International Conference on Distributed Computing Systems*, pages 186–192, Washington D.C., 1979. IEEE Computer Society.
- [17] P. Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford. T Spaces. *IBM Journal of Research and Development*, 37(3 - Java Techonology):454–474, 1998.