



ReSpecT Guide

ReSpecT version: 1.0.0

Document revision: 001
Creation date: 2008-04-05
Last Changes date: 2008-04-05

DEIS, Università di Bologna, Italy
Sede di Cesena (FC)
Sede di Bologna (BO)

Contents

I	ReSpecT Basics	3
1	Introduction	4
2	Getting started	7
3	ReSpecT IDE	8
3.1	The CLIAgent tool	8
3.2	The Inspector tool	8
4	ReSpecT API for ReSpecT Tuple Centres from Java Agents	9
4.1	Accessing ReSpecT from Java	9
II	ReSpecT Advanced	10
5	Programming Tuple Centre Behaviour	11
5.1	The ReSpecT Language	11
5.2	Coordination as computations on the interaction space	13
5.2.1	Basic computation constructs in ReSpecT: Selection, Iteration and Recursion . .	14
5.3	Common mistakes	16
5.3.1	Binding variables more than once	16
III	Engineering systems with ReSpecT	19
6	Examples and Patterns	20
6.1	Some Communication Patterns	20
6.1.1	Blackboard-based Communication	20
6.1.2	Message-oriented Communication	21
6.2	Simulating New Coordination Primitives	21
6.2.1	rd_all	22
6.2.2	in_all	22
6.3	Basic Synchronisation Patterns	23
6.3.1	Semaphores	23
6.3.2	Synchronisation Barrier	24
6.4	Other Basic Coordination Patterns	25
6.4.1	Interoperability	25
6.4.2	Resource Allocation	25
6.5	Some Coordination Idioms	25
6.5.1	Tuple on demand	26
6.6	Other examples	26

6.6.1	Dining Philosophers	26
6.6.2	Contract Net Protocol	26
6.6.3	Workflow Engine	26
IV	Extensions	27
7	Timed ReSpecT	28
7.1	Introduction	28
7.2	Extending ReSpecT with Time	28
7.3	Examples	31
7.3.1	Timed Requests	32
7.3.2	Tuples in Leasing	33
7.3.3	Dining Philosophers with Maximum Eating Time	34
7.3.4	An Artifact for Timed Contract Net Protocols	37
A	Related Documents and Publications	41
B	Document History	42
C	Planned Development of this Document	43

Part I

ReSpecT Basics

Chapter 1

Introduction

ReSpecT (Reaction Specification Tuples¹) is a logic-based language for MAS coordination [9]. ReSpecT is based on a coordination model providing *tuple centres* as first-class abstractions to design and develop general-purpose coordination media. The behaviour of ReSpecT tuple centres is programmed through the ReSpecT first-order logic language. A tuple centre is a tuple space enhanced with the possibility to program its behaviour in response to interactions. So, first of all, agents can operate on a ReSpecT tuple centre in the same way as on a Linda tuple space [5]: by exchanging *tuples* (which are ordered collection of knowledge chunks) through a simple set of coordination primitive. An agent can write a tuple in a tuple centre with an *out* primitive; or read a tuple from a tuple centre with primitives such as *in*, *rd*, *inp*, *rdp* specifying a *tuple template*—that is, an identifier for a set of tuples, according to some *tuple matching* mechanism. Reading tuples can be destructive (*in*, *inp* remove the matching tuple) or non-destructive (*rd*, *rdp* simply read the matching tuple), suspensive (*in*, *rd* wait until a matching tuple is found) or non-suspensive (*inp*, *rdp* immediately return either the matching tuple or a failure result)—but is anyway non-deterministic: when more than one tuple in a tuple centre are found that match a tuple template, one is non-deterministically chosen among them.

Accordingly, a tuple centre enjoys all the many features of a tuple space, which can be classified along three different dimensions: generative communication, associative access, and suspensive semantics. The main features of generative communication (where information generated has an independent life with respect to the generator) are the forms of uncoupling (space, time, name) based on mediated interaction: sender and receiver do not need to know each other, to coexist in the same space or at the same time in order to communicate (to exchange a tuple, in particular). Associative access (access based on structure and content of information exchanged, rather than on location, or on name) based on tuple matching promotes synchronisation based on tuple structure and content: thus, coordination is data-driven, and allows for knowledge-based coordination patterns. Suspensive semantics promotes coordination patterns based on knowledge availability, and couples well with incomplete or partial knowledge.

A tuple centre is a programmable tuple space—thus adding programmability of the coordination medium as a new dimension of coordination. While the behaviour of a tuple space in response to interaction events is fixed (so, the effects of coordination primitives is fixed), the behaviour of a tuple centre can be tailored to the application needs by defining a set of specification tuples, or reactions, which determine how a tuple centre should react to incoming / outgoing events.

While the basic tuple centre model is not bound to any specific language to define reactions [9], ReSpecT tuple centres are obviously programmed through the ReSpecT logic-based specification language. The original ReSpecT [8] is a logic-based language for the specification of the behaviour of tuple centre. As a behaviour specification language, ReSpecT:

- enables the definition of computations within a tuple centre, called *reactions*, and
- makes it possible to associate reactions to events occurring in a tuple centre.

¹The ReSpecT technology is available as an open source project from the ReSpecT web site [1]

So, ReSpecT has both a declarative and a procedural part. As a *specification language*, it allows events to be declaratively associated to reactions by means of specific logic tuples, called *specification tuples*, whose form is $\mathbf{reaction}(E, R)$. In short, given an event Ev , a specification tuple $\mathbf{reaction}(E, R)$ associates a reaction $R\theta$ to Ev if $\theta = mgu(E, Ev)$.² As a reaction language, ReSpecT enables reactions to be procedurally defined in terms of sequences of logic reaction goals, each one either succeeding or failing. A reaction as a whole succeeds if all its reaction goals succeed, and fails otherwise. Each reaction is executed sequentially with a transactional semantics: so, a failed reaction has no effect on the state of a logic tuple centre.

All the reactions triggered by an event are executed before serving any other event: so, agents perceive the result of serving the event and executing all the associated reactions altogether as a single transition of the tuple centre state. As a result, the effect of a coordination primitive on a logic tuple centre can be made as complex as needed by the coordination requirements of a system. Generally speaking, since ReSpecT has been shown to be Turing-equivalent [3], any computable coordination law could be in principle encapsulated into a ReSpecT tuple centre. This is why ReSpecT can be assumed as a general-purpose core language for coordination: a language that could then be used to represent and enact policies and rules for coordination systems of any sort.

Adopting the declarative interpretation of logic tuples, a ReSpecT tuple centre has then a twofold nature [8]: a theory of communication (the set of the ordinary tuples) and a theory of coordination (the set of the specification tuples). This allows in principle intelligent agents to reason about the state of collaboration activities, and to possibly affect their dynamics. Furthermore, the twofold interpretation of ReSpecT specification tuples (either declarative or procedural) allows knowledge and control to be represented uniformly (as Prolog-like facts) and encapsulated within the same coordination artifact.

² mgu is the most general unifier, as defined in logic programming.

Chapter 2

Getting started

TO BE DONE

Chapter 3

ReSpecT IDE

3.1 The CLIAgent tool

TO BE WRITTEN DOWN

3.2 The Inspector tool

TO BE DEVELOPED: MAYBE BY A THESIS?

Chapter 4

ReSpecT API for ReSpecT Tuple Centres from Java Agents

4.1 Accessing ReSpecT from Java

Part II

ReSpecT Advanced

Chapter 5

Programming Tuple Centre Behaviour

As pointed out in Chapter 1, tuple centres are programmable tuple spaces whose behaviour can be programmed by using ReSpecT as a logic-based coordination language to define 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 [8].

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_tuple(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)))
```

<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 ReSpecT

Chapter 6

Examples and Patterns

In this chapter we present a set of simple examples and patterns, where ReSpecT is adopted as a coordination language to realise agent communication and coordination. The examples can be considered samples for a 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.

6.1 Some Communication Patterns

6.1.1 Blackboard-based Communication

Obviously, blackboard based communication is supported naturally in ReSpecT. 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.

6.1.2 Message-oriented Communication

ReSpecT 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;

6.2 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.

6.2.1 `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])),
```

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

```
out_r(H),
out_r(rdall_restore(T)) )).
```

6.2.2 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)))).
```

6.3 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 encapsulating 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.

6.3.1 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.

6.3.2 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).

6.4 Other Basic Coordination Patterns

6.4.1 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

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

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.

6.4.2 Resource Allocation

6.5 Some Coordination Idioms

Coordination idioms are frequent solution adopted exploiting the peculiar features of the ReSpecT coordination model.

6.5.1 Tuple on demand

6.6 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).

6.6.1 Dining Philosophers

6.6.2 Contract Net Protocol

6.6.3 Workflow Engine

Part IV

Extensions

Chapter 7

Timed ReSpecT

7.1 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, this version 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.

7.2 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 [10, 8] — 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 [8]. 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.

7.3 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.

7.3.1 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 7.1 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 7.1: 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 7.2: 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 7.3: ReSpecT specification mimicking an `inall` with a duration time

7.3.2 Tuples in Leasing

In this example we model the notion of *lease*, analogously to the lease notion in models such as JavaSpaces [4] and TSpaces [14]. 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 7.2 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).

7.3.3 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 [10]).

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:

```
while (true){
```

```

% 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 7.4: ReSpecT specification for coordinating dining philosophers with a maximum eating time

```

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

```

The coordination specification in ReSpecT (first 6 reactions of Table 7.4, 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 7.4 (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.

7.3.4 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 bulding more articulated protocols and coordination strategies [13]. Following [6], 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 7.6. Table 7.5 shows the pseudo-code representing the interactive behaviour of the master (top) and workers (bottom).

```

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 7.5: Sketch of the behaviour of the agents participating to the timed Contract Net Protocol: masters (*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 7.6 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(out_r(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 7.6: Behaviour of the artifact realising a timed CNP, encoded in the ReSpecT language

Appendix A

Related Documents and Publications

The updated list of articles, papers and documents about ReSpecT can be found at the official ReSpecT website

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

- [10] – describes the tuple centre coordination model;
- [8] – 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;
- [11] – introduces TuCSon as Coordination Model/Infrastructure for Internet Application Development;
- [7] – describes the concept of hybrid coordination model;
- [12] – book about coordination models, technologies and applications for Internet agents;

Appendix B

Document History

Date	Action	Responsible
2008-04-05	The document is created	macasadei

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 the concept of ReSpecT tuple centre as a means to build concurrent systems on a single node
- the relations between TuCSoN and ReSpecT: TuCSoN wraps a ReSpecT tuple centre by adding new services and putting it in a distributed context.
- the twofold nature of ReSpecT as a coordination language and a technology to actually develop concurrent systems.
- Description of the entire ReSpecT Java API.
- Description of the Java tools for inspecting ReSpecT tuple centres and related coordination laws.
- Of course, development of the previously mentioned tools.

Bibliography

- [1] aliCE Research Group. ReSpecT home page. <http://www.alice.unibo.it/xwiki/bin/view/ReSpecT/>.
- [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, USA, 27 Feb.– 1 Mar. 1998. ACM. Special Track on Coordination Models, Languages and Applications.
- [4] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces: Principles, Patterns, and Practice*. The Jini Technology Series. Addison-Wesley, 1999.
- [5] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
- [6] 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.
- [7] A. Omicini. Hybrid coordination models for handling information exchange among Internet agents. In A. Bonarini, M. Colombetti, and P. L. Lanzi, editors, *Workshop “Agenti intelligenti e Internet: teorie, strumenti e applicazioni”*, pages 1–4, 7th AI*IA Convention (AI*IA 2000), Milano, Italy, 13 Sept. 2000.
- [8] A. Omicini and E. Denti. Formal ReSpecT. *Electronic Notes in Theoretical Computer Science*, 48:179–196, June 2001. Declarative Programming – Selected Papers from AGP 2000, La Habana, Cuba, 4–6 Dec.2000.
- [9] A. Omicini and E. Denti. From tuple spaces to tuple centres. *Science of Computer Programming*, 41(3):277–294, Nov. 2001.
- [10] A. Omicini and E. Denti. From tuple spaces to tuple centres. *Science of Computer Programming*, 41(3):277–294, Nov. 2001.
- [11] 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.
- [12] A. Omicini, F. Zambonelli, M. Klusch, and R. Tolksdorf, editors. *Coordination of Internet Agents: Models, Technologies, and Applications*. Springer-Verlag, Mar. 2001.

- [13] 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.
- [14] 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.