Timed Coordination Artifacts with ReSpecT

Alessandro Ricci
DEIS
Università di Bologna – Sede di Cesena
via Venezia 52, 47023 Cesena (FC), Italy
Email: aricci@deis.unibo.it

Mirko Viroli
DEIS
Università di Bologna – Sede di Cesena
via Venezia 52, 47023 Cesena (FC), Italy
Email: mviroli@deis.unibo.it

Abstract— **Environment-based approaches to Multi-Agent Systems (MAS) advocate the use of abstractions mediating the interaction between agents, providing an alternative viewpoint to the standard speech-act-based approach. A remarkable example is rooted in the notion of *coordination artifact*: embodied entities provided by the MAS infrastructure to automate a specific coordination task, and featuring peculiar engineering properties such as encapsulation, predictability, inspectability and malleability. An example technology supporting this scenario is TuCSoN, where coordination artifacts are built as tuple centres programmed with the ReSpecT logic language.**

In most application scenarios characterised by a high degree of openness and dynamism, coordination tasks need to be time-dependent so as to be able to specify and guarantee necessary levels of liveness and of quality of service. Moreover, temporal properties are also fundamental for intercepting violations in the agent-artifact contract, which is at the root of the engineering approach underlining coordination artifacts. Accordingly, in this paper we introduce an extension to the ReSpecT language allowing to define timed coordination artifacts in the TuCSoN infrastructure. This is achieved by adding the management of trap events, fired and intercepting using the same mechanism currently used by ReSpecT to handle communication events, thus in a uniform and coherent way. Examples are provided to show the expressiveness of the language to model temporal-based coordination tasks.

I. Introduction

In the context of Environment-based approaches to interaction on Multi-Agent Systems (MAS), the notion of *coordination artifact* has been introduced as the root of an engineering methodology for agent coordination [10], [15]. The key idea of this approach is to equip the MAS with a coordination infrastructure, providing abstractions — called coordination artifacts — perceived by the agents as run-time entities living in the environment. Coordination artifacts are designed with the goal of automating a specific coordination task, provided to the agents as a service, and featuring peculiar engineering properties such as encapsulation, predictability, inspectability and malleability [10], [15].

An example technology supporting this scenario is TuCSoN [11], [14]. In TuCSoN, the nodes of the network can be populated by *tuple centres* playing the role of coordination artifacts. Tuple centres are LINDA-like blackboards, whose reactive behaviour can be programmed using the logic-based language ReSpecT, so as to make the tuple centres encapsulating any coordination task, from simple synchronization policies up to complex workflows. In particular, ReSpecT is shown to be Turing-complete, thus allowing any coordination algorithm to be specified.

However, in most application scenarios characterised by a high degree of opennes 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. As shown in [15], it is sensible e.g. to let coordination artifacts provide agents with operating instructions featuring timed properties, which can be correctly enforced only through timed coordination tasks.

The need for specifying timed coordination policies emerged in a parallel way in the field of distributed systems as well. For instance, in JavaSpaces [4] primitives `read` and `take` — looking for a tuple analogously to `rd` and `in` in LINDA — comes with a timeout value: when the timeout expires the request immediately returns a failure. Similarly, tuples can provide a *lease* time when inserted in the space: when the lease expires the tuple is automatically removed. All these primitives, and others based on time, can actually be the basis for structuring more complex coordination scenarios, such as e.g. auctions and negotiations protocols including time-based guarantees and constraints.

In this work we discuss how the basic ReSpecT tuple centre model has been extended to support the definition and enaction 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.

The rest of the paper is organised as follows: Section II discusses in details the ReSpecT extended model, Section III provides some concrete examples exploiting the extended model, Section IV provides some reflections on the features of the approach and finally Section V provides related works, conclusion and future works.

M. Baldoni, F. De Paoli, A. Martelli, and A. Omicini (Eds.): *WOA 2004 – Dagli Oggetti agli Agenti, Sistemi Complessi e Agenti Razionali*, Torino, Italy, November 29th – December 1st 2004. Pitagora Editrice Bologna. ISBN 88-371-1533-4. `http://woa04.unito.it`

σ	::=	{`reaction(`*p*`(t),(` *body* `)).`}	Specification
p	::=	*cp* | *rp*	ReSpecT primitives
cp	::=	`out` | `in` | `rd`	Comm. primitives
rp	::=	`in_r` | `rd_r` | `out_r` | `no_r`	Reaction primitives
body	::=	[*goal*{`,`*goal*}]	Specification body
ph	::=	`pre` | `post`	Direction predicates
goal	::=	*ph* | *rp*`(t)`	Goals

Fig. 1. The syntax of a ReSpecT specification

II. Extending ReSpecT with Time

We describe here informal semantics of a significant fragment of the ReSpecT language: the reader interested in a formal presentation should refer to [9], [8]. Then, we describe how this model can be extended so as to deal with timing aspects, that is, with the ability to trigger trap events at a specified time (in the future).

A. The Basic Model

ReSpecT [8] is a logic-based language to program the reactive behaviour of tuple centres [9].

Tuple centres are *coordination media* extending the basic model of LINDA tuple spaces [5]. Similarly to LINDA, they accept and serve requests for inserting a tuple `t` (by primitive `out(t)`), removing a tuple matching template `tt` (by primitive `in(tt)`), and reading a tuple matching template `tt` (by primitive `rd(tt)`)[1]. With respect to LINDA, ReSpecT tuple centres specialise the tuple space model with logic tuples (Prolog-like terms with variables) and unification as the matching criterion; differently from LINDA tuple spaces, tuple centres can be programmed so that whenever an external communication event occurs a computation reactively starts which may affect the state of the inner tuple space. External communication events can either be *(i)* a *listening*, reception of a request from a coordinated process (either a `in`, `rd`, `out`), or *(ii)* a *speaking*, the production of a reply towards a coordinated process (either the reply to a `in` or `rd`)[2].

The ReSpecT language can be used to declare a set σ of *reaction specification tuples* (RSTs), using the syntax of Figure 1.

Each RST has a head and a body. When a communication event *p*`(t)` occurs, all the RSTs with a matching head are activated, that is, their bodies — each specifying an atomic computation over the tuple centre — are used to spawn a pending reaction waiting to be executed. Being specified by a body, reactions are composed by a sequence of reaction primitives *rp* resembling LINDA primitives, which are used to remove a tuple (`in_r`), read a tuple (`rd_r`), insert a tuple (`out_r`), and check for the absence of a tuple (`no_r`). This sequence can contain a direction predicate *ph*, `pre` or `post`, which is used to filter between reactions to a listening or a speaking. In particular, we here consider therefore five kinds of external communication events: listening of a `out`, `rd`, or `in`, and speaking of a `in` and `rd`.

Reactions are non-deterministically picked and executed, by atomically executing all its reaction primitives. Their effect is to change the state of the tuple centre, and to fire new reactions, as long as they match some other RST — whose head can specify a reaction primitive (internal communication events) other than a communication primitive (external communication events). This recursive creation of reactions is the mechanism by which ReSpecT achieves expressiveness up to reaching Turing-completeness [3].

Primitives `in_r`, `rd_r`, and `no_r` might fail (the former two when the tuple is absent, the latter when it is present), in which case the reaction execution fails, and its effect on the tuple centre is rolled back. The computation fired by the external communication event stops when (if) no more pending reactions occur: when this happens the tuple centre waits until the next communication event occurs.

B. The Extended Model

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 time [3]. Actually, tuple centre time is a physical time, but it is value 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*`)`[4]

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.

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

[1]Tuple centres can also deal with usual predicative primitives `inp(tt)` and `rdp(tt)` of LINDA, but these are not considered here for the sake of simplicity and without loss of generality.

[2]we use here the term *listening* related to events following the basic terminology adopted in [9]

[3]In current implementation the temporal unity is the millisecond

[4]A 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

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 [9], [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.

As will be discussed in Section IV, the possibily 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 installed[5].

III. 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

[5]This 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 I
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 II
ReSpecT SPECIFICATION FOR MODELLING TUPLES WITH A LEASE TIME

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.

A. 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 I 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).

B. 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 [16]. 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 II 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 `outl` 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 `outl` tuple carrying information about the presence of the leased tuple is removed (reaction 5).

C. 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 [9]).

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 provides back the chopsticks by inserting the tuple `chops(C1,C2)` in the tuple centre, and *(iv)* finally starts thinking until next dining cycle.

A pseudo-code reflecting this interactive behaviour is the following:

```
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 III
ReSpecT SPECIFICATION MIMICKING AN INALL WITH A DURATION TIME

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

The coordination specification in ReSpecT (first 6 reactions of Table IV, 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 IV (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 fime (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

```
% 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 IV
ReSpecT SPECIFICATION FOR COORDINATING DINING PHILOSOPHERS WITH A MAXIMUM EATING TIME

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.

D. 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 VI. Table V 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 insering the tuple task_result).

The artifact behaviour in ReSpecT described in Table VI 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) awared 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 concening 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).

```
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 V
SKETCH OF THE BEHAVIOUR OF THE AGENTS PARTICIPATING TO THE TIMED CONTRACT NET PROTOCOL: MASTERS *(Top)* AND WORKERS *(Bottom)*

IV. DISCUSSION

The approach aims to be general and expressive enough to allow the description of a large range of coordination patterns based on the notion of time. An alternative way to solve

```
% 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 VI
BEHAVIOUR OF THE ARTIFACT REALISING A TIMED CNP, ENCODED IN THE ReSpecT LANGUAGE

the problem consists in adopting helper agents (sort of *Timer* agents) with the specific goal of generating traps by inserting specific tuples in the tuple centre a certain time points. With respect to this approach and also to other approaches, the solution described in this work has several advantages:

- *Incapsulation of coordination* — Managing traps directly inside the coordination medium makes it possible to fully keep coordination encapsulated, embedding its full specification and enactment in a ReSpecT program and tuple centre behaviour. Conversely, using helper agents to realise part of the coordination policies which cannot be expressed directly in the medium causes a violation of encapsulation. Among the problems that arise, we have: less degree of control, more problematic reusability and extensibility, more complex formalisation.
- *Timed-coordination* — The approach is not meant to provide strict guarantees as required for real time systems: actually, this would be difficult to achieve given also the complexity of ReSpecT behaviours, based on first order logic. However, the model is expressive and effective enough to be useful for several kind of timed systems in general. Also, the management of time events directly inside the medium makes it possible to have some guarantees on the timings related to trap generation and

trap reaction execution. These guarantees would not be possible in general adopting external agents simulating traps by inserting tuples at (their) specific time. The reacting stage of a tuple centre has always priority with respect to listening of communication events generated by external agents; this means that in the case of complex and articulated reaction chains, the listening of a trap event (i.e. reacting to tuples inserted by timer agents) could be substantially delayed, and possibly could not happen. On the contrary, this cannot happen in the extended model, where a trap event is ensured to be listened and the related reactions to be executed — with higher priority.

- *Well-founded semantics* — The extension realised to the basic model allows for a well-defined operational semantics extending the basic semantics of tuple centres and ReSpecT with few constructs and behaviours. In particular, the basic properties of ReSpecT – in particular atomic reaction execution – are all preserved. This semantics has been fundamental for driving the implementation of the model and will be important also for the development of verification tools.
- *Compatibility, reuse and minimality* — The extension does not alter the basic set of (Linda) coordination primitives, and then it does not require learning and adopting new interfaces for agents aiming to exploit it: all the new features are at the level of the coordination medium programming. This in particular implies that the new model can be introduced in existing systems, exploiting the new temporal features without the need to change existing agents.
- *"The hills are alive"* — Coordination artifacts with temporal capabilities can be suitably exploited to model and engineer *living environments*, i.e. environments which spontaneously change with some kind of transformations, due to the passage of time. A well known example is given by environments in the context of stigmergy coordination approaches with multi-agent systems [12]; in this context, the pheromones (part of the agents – ants – environment) evaporate with the passing of time according to some laws which heavily condition the emerging coordination patterns. Tuple centres can be exploited then to model and enact the living environment: tuples can represent pheromones (placed to and perceived from the environment by mean of the basic coordination primitives), and tuple centre behaviour can embed the rules describing how to transform pheromones with the passage of time.

Concerning the implementation of the model, the tuple centre centralisation vs. distribution issue arises. The basic tuple centre model is not necessarily centralised: however, the extension provided in this work — devising out a notion of time for each medium — leads quite inevitably to realise tuple centres with a specific spatial location. This is what already happens in TuCSoN coordination infrastructure, where there can be multiple tuple centres distributed over the network, collected and localised in infrastructure nodes. It is worth mentioning that this problem is not caused by our framework, but is inherent on any approach aiming at adding temporal aspects to a coordination model.

However, according to our experience in agent based distributed system design and development, the need to have a distributed implementation of individual coordination media is a real issue only for very specific application domains. For the most part of applications, the bottleneck and single point of failure arguments against the use of centralised coordination media can be answered by a suitable design of the multi-agent system and an effective use of the coordination infrastructure. At this level, it is fundamental that a software engineer would know the scale of the coordination artifacts he is going to use, and the quality of service (robustness in particular) provided by the infrastructure.

V. Related Works and Conclusion

The contribution provided by this work can be generalised from tuple centre to — more generally — the design and development of general purpose time-aware coordination artifacts in multi-agent systems [10].

Outside the specific context of coordination models and languages, the issue of defining suitable languages for specifying the communication and coordination in (soft) real time systems have been studied for long time. Examples of such languages are Esterel [1] and Lustre [2], both modelling synchronous systems, the former with an imperative style, and the latter based on dataflow. In coordination literature several approaches have been proposed for extending basic coordination languages with timing capabilities. [7] introduces two notions of time for Linda-style coordination models, relative time and absolute time, providing different kind of features. Time-outs have been introduced in JavaSpaces [4] and in TSpaces [16].

The approach described in this work is quite different from these approaches, since it extends the basic model without altering the basic Linda model from the point of view of the primitives, but acting directly on the expressiveness of the coordination media. Also, it does not provide specific time capabilities, but — following the programmable coordination media philosophy — aims at instrumenting the model with the expressiveness useful for specifying any time-based coordination pattern.

The model has been implemented in the version 1.4.0 of TuCSoN coordination infrastructure, which is available for downloading at TuCSoN web site [14]. Ongoing work is concerned with defining a formal operational semantics of the extended model, consistent and compatible with the basic one defined for ReSpecT [9], [8]. The formal semantics is important in particular to frame the expressiveness of the model compared to existing models in literature concerned with timed systems, and to explore the possibility of building theories and tools for the verification of formal properties.

Future work will stress the approach with the engineering of real world application domain involving time in the coordination activities.

REFERENCES

[1] G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[2] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: a declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 178–188. ACM Press, 1987.

[3] E. Denti, A. Natali, and A. Omicini. On the expressive power of a language for programming coordination media. In *Proc. of the 1998 ACM Symposium on Applied Computing (SAC'98)*, pages 169–177. ACM, February 27 - March 1 1998. 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, January 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] J.-M. Jacquet, K. De Bosschere, and A. Brogi. On timed coordination languages. In D. Garlan and D. Le Métayer, editors, *Proceedings of the 4th International Conference on Coordination Languages and Models*, volume 1906 of *LNCS*, pages 81–98, Berlin (D), 2000. Springer-Verlag.

[8] 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.

[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, A. Ricci, M. Viroli, C. Castelfranchi, and L. Tummolini. Coordination artifacts: Environment-based coordination for intelligent agents. In N. R. Jennings, C. Sierra, L. Sonenberg, and M. Tambe, editors, *3rd international Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004)*, volume 1, pages 286–293, New York, USA, 19–23 July 2004. ACM.

[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] V. D. Parunak. 'Go To The Ant': Engineering principles from natural agent systems. *Annals of Operations Research*, 75:69–101, 1997.

[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] TuCSoN home page. `http://lia.deis.unibo.it/research/TuCSoN/`.

[15] M. Viroli and A. Ricci. Instructions-based semantics of agent mediated interaction. In N. R. Jennings, C. Sierra, L. Sonenberg, and M. Tambe, editors, *3rd international Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004)*, volume 1, pages 102–110, New York, USA, 19–23 July 2004. ACM.

[16] 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.