Time-Aware Coordination in ReSpecT

Andrea Omicini, Alessandro Ricci, and Mirko Viroli

DEIS, Alma Mater Studiorum, Università di Bologna,
via Venezia 52, 47023 Cesena, Italy
{andrea.omicini, a.ricci}@unibo.it, mviroli@deis.unibo.it

Abstract. Tuple centres allow for dynamic programming of the coordination media: coordination laws are expressed and enforced as the behaviour specification of tuple centres, and can change over time. Since time is essential in a large number of coordination problems and patterns (involving timeouts, obligations, commitments), coordination laws should be expressive enough to capture and govern time-related issues.

Along this line, in this paper we discuss how tuple centres and the ReSpecT language for programming logic tuple centres can be extended to catch with time, and to support the definition and enforcement of time-aware coordination policies. Some examples are provided to demonstrate the expressiveness of the ReSpecT language to model timed coordination primitives and laws.

1 Introduction

Coordination artifacts are general-purpose run-time abstractions embedded in a MAS (multi-agent system) coordination infrastructure [1, 2], and meant to provide agents with *coordination as a service* [3]. In particular, coordination artifacts aim at automating specific coordination tasks, encapsulated outside the agents, and featuring relevant engineering properties such as predictability, inspectability and malleability of behaviour [1].

The reference coordination model supporting the notion of coordination artifact is TuCSoN [4, 5]. TuCSoN *tuple centres* populate network nodes and play the role of coordination artifacts. Tuple centres are LINDA-like tuple spaces [6], whose reactive behaviour can be programmed using the logic-based, Turing-complete language ReSpecT [7]. By this language, tuple centres can encapsulate any coordination task, from simple synchronisation policies up to complex workflows [8, 9].

However, in most application scenarios characterised by a high degree of openness and dynamism, coordination tasks need to be time dependent. On the one hand, handling time is necessary to specify (and enforce) given levels of liveness and quality of service: for instance, an agent could be required to interact with a coordination artifact at a given minimum / maximum frequency. On the other hand, temporal properties are also fundamental in the agent-artifact contract: for instance, an agent could commit to accomplish a task before a given timeout expires, or could require the artifact to provide a response within a given time.

J.-M. Jacquet and G.P. Picco (Eds.): COORDINATION 2005, LNCS 3454, pp. 268–282, 2005.

The expressive need for timed coordination policies already emerged in the field of distributed systems as well. For instance, in JavaSpaces [10] primitives `read` and `take` — looking for tuples in the same way as `rd` and `in` in LINDA — come with a timeout value: when the timeout expires without a matching tuple is found, a failure result is returned. Similarly, tuples can be equipped with a *lease* time when they are inserted in the space: as soon as the lease expires, the tuple is automatically removed. All these primitives, and others based on time, can actually be used as the basis for structuring more complex coordination scenarios, such as e.g. auctions and negotiations protocols including time-based guarantees and constraints. However, they are typically too specific solutions to capture all the time-related coordination problems, which instead require a general and comprehensive model for time in coordination.

Along this line, in this paper we discuss how the basic ReSpecT tuple-centre model has been extended to support the definition and enaction of time-aware coordination policies. Section 2 discusses the general concept of time-aware coordination artifacts, and describes how tuple centres and ReSpecT can be extended accordingly to deal with time. Section 3 exemplifies the approach by showing how the dining-philosopher problem can be modelled in ReSpecT, and then extended with time constraints. Section 4 briefly discusses the expressiveness of the extended ReSpecT tuple-centre model, by showing how it can be used to express some well-known temporal features as found in a number of well-known coordination models (such as tuple leasing and timed requests). Section 5 gives some clues of the model implementation. Finally, Section 6 considers related works, and Section 7 provides for conclusions.

2 Timed Coordination Artifacts

In order to represent and enforce timed coordination laws within a coordination artifacts, some conceptual and practical pre-conditions have to be satisfied, and some issues need to be properly addressed.

First of all, time has to be an integral part of the ontology of a coordination artifact. Generally speaking, time can be local or global (if it refers to the global system time, or to the local artifact time), relative or absolute (if it assumes as zero the starting time of the artifact, or uses a standard time convention like date-time), continuous or discrete. Typically, local and relative time are the most natural reference for coordination artifacts in distributed systems, since it can be always be defined and used with no conceptual difficulties. Global and absolute time can be defined conventionally / pragmatically from there — for instance, global system time is the time of a specific artifact (along with some practical methods to extract time from there, and define some notion of simultaneity), absolute time is obtained by properly labelling time 0 of the artifact, and then using relative time as a delta. Finally, discrete time is the obvious choice for any computational machine. As a result, a coordination artifact can label any event (either incoming or outgoing) with its own (local, relative and discrete) time, which is then amenable to be used as a unique label within the coordination

artifact, under the simple hypotheses that it works with a single flow of control, and has a fine-enough-grained time scale.

Correspondingly, a coordination artifact should allow coordination laws to talk about time. A range of predicates / functions has then to be provided as the syntactic sugar to access the time information (label) featured by any event to be handled (the time when the action that produced the event has affected the artifact, and the current time of the computation within the artifact), and to perform simple computations over time (comparing time points and / or intervals).

Of course, time has to be embedded into the coordination-artifact working cycle. So, some notion of *time event* has to be introduced, which triggers some time-related computation within the artifact. In fact, it is not enough to allow the time of an event to be accessed: time-related laws like "the answer should come within 3 minutes after the question, otherwise some penalty will be inflicted" cannot be expressed only referring to actions actually performed — since they contemplate the case of no action performed, the corresponding law cannot refer to a reaction to no action. So, time events (as events triggering some behaviour of the coordination artifact, conceptually corresponding to the passage of time) have to be autonomously generated by the coordination artifact in order to suitably handle time-related laws, within the normal working cycle of the artifact. Quite obviously, attention should be paid on the one hand not to overload the cycle, on the other hand not to be too coarse in time intervals.

Along this line, the ability to capture time events and to react appropriately is another obvious capability required to a timed coordination artifact. Time-aware coordination laws can then be enforced, that can specify / constraint behaviours that depend on time, and can suitably relate time with the evolution of the coordination (artifact) state.

Finally, talking about time naturally recalls the dynamics that is typically featured by the coordination laws encapsulated within coordination artifacts. So, a fundamental complement to the ability of the artifact to specify and enact time-related coordination policies is the ability to modify the coordination specification over time, during the "active life" of the artifact, and possibly depending on some time-aware behaviour of the artifact itself. Correspondingly, it should be possible to express how to add a new coordination law, and how to remove an old one, so as to adapt the artifact behaviour (and the coordination altogether) to the passage of time — or to the change, more generally.

2.1 Timed Tuple Centres

Tuple centres are introduced in [7] as coordination artifacts meant at engineering coordination activities in MASs. Technically, a tuple centre is a *programmable* tuple space, i.e. a tuple space whose reactive behaviour to communication events can be programmed so as to specify and enact any coordination policy [7]. Tuple centres can be thought then as general purpose coordination artifacts, which can be suitably forged in order to provide specific coordination services. The tuple centre model is not bound to any specific model / language for behaviour spec-

ification or to a specific communication language: these aspects are defined by specific instances of the model. An example, discussed in next subsection, is given by ReSpecT tuple centres [11], which adopt logic tuples as the communication language, and the ReSpecT language for tuple centre behaviour specification. Independently of the specific language adopted, the tuple centre behaviour is meant to be specified in terms of reactions to (communication) events occurring in the artifact. So, the core idea behind tuple centres (and coordination artifacts, more generally) is to have first-class coordination abstractions which are powerful enough to encapsulate and enforce at execution time the coordination laws required to support MAS activities. This does not happen, for instance, in basic LINDA-like models, where complex coordination activities surpassing the limited expressive power of tuple space coordination force the global logic of coordination to be spread among individual agents [7]. As coordination artifacts, tuple centres have a usage interface, composed by the basic LINDA primitives, plus two primitives — `set_spec` and `get_spec` — for setting and reading the tuple centre behaviour specification. As coordination artifacts, tuple centres also feature inspectability and malleability properties, i.e. their coordinating behaviour can be inspected and changed dynamically, at execution time.

Timed tuple centres extend tuple centres with the temporal framework depicted above for timed coordination artifacts. First, the notion of *current time* for a tuple centre is introduced as a local, relative and discrete time. Conceptually, tuple-centre time is generated by an inner clock owned by the tuple centre: no relationships can be established in principle between the current time of two different tuple centres. The current time of a timed tuple centre is zero when the tuple centre is actually created by the infrastructure at run time. Absolute time is available, conventionally computed by suitably adding the (absolute) tuple-centre creation time (as provided by the infrastructure) to the current time.

With respect to the formal model defined in [7], a *time transition* is introduced in the basic tuple centre working cycle, in addition to the existing *listening*, *speaking*, and *reacting* transitions. The time transition is meant to have priority with respect to all the other transitions, including the reacting one. Conceptually, the time transition is executed at each tick of the tuple centre clock — as reacting to the generation of a *time event* for each tick.[1]

Then, similarly to communication events, it is possible to specify reactions triggered by time events (*timed reactions*). Timed reactions follow the same semantics of other reactions: once triggered, they are placed in the triggered-reaction set and then executed, atomically, in a non-deterministic order. Since at a given time, only one time event can occur, each timed reaction is executed only once.

As a result, a timed tuple centre can be programmed to react to the passing of time, so as to enforce time-aware coordination policies.

[1] In practice, then, the time transition needs to be executed only when the tuple centre specification actually contains triggerable timed reactions, according to a simple mechanism sketched in Section 5.

2.2 Timed ReSpecT

ReSpecT tuple centres are tuple centres based on first-order logic, adopted both for the communication language (logic tuples), and for the behaviour specification language (ReSpecT) [11]. Basically, reactions in ReSpecT are defined as Prolog-like relations of the form

```
reaction(Head, Body).
```

which specify the list of the operations to be executed (the *Body* of the reaction) when a certain communication event occurs (represented by the reaction *Head*). Such operations make it possible to inspect and change current communication and coordination state, for instance by inserting / reading / removing tuples from the tuple set (see [11] for details). Operations can trigger new reactions. If just only one of the operations invoked in the body of the reaction fails, the entire reaction fails atomically, rolling back any change possibly done by previous operations successfully executed by the same reaction.

According to the timed-tuple-centre model described in Subsection 2.1, the ReSpecT language is extended with time *(i)* by introducing some temporal predicates to get information about both tuple-centre and event time, and *(ii)* by making it possible to specify reactions on the occurrence of time events. The temporal predicates introduced are the following:[2]

- `current_time(?Time)` This predicate succeeds if *Time* (typically a variable) unifies with the current tuple-centre time. As an example, the reaction specification tuple

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

  inserts a new tuple (`request_log`) with timing information each time a request to retrieve a tuple `p(X)` is executed, thus implementing the temporal log of a specific sort of request.
- `event_time(?Time)` This predicate succeeds if *Time* unifies with the tuple-centre time when the original communication event triggering the reaction occurred.
- `before(@Time)`, `after(@Time)`, `between(@MinTime,@MaxTime)` These predicates succeeds if the current tuple-centre time is respectively less than, greater than, and between the specified temporal arguments.

Reactions to time events are specified analogously to ordinary reactions:

```
reaction(time(Time), Body).
```

where *Time* is a ground term. The intended semantics is the following: as soon as the tuple-centre time reaches the *Time* value — so, the time event `time(Time)` is conceptually generated — all the reactions whose head matches the event time can be triggered, and their *Body* inserted in the triggered-reaction set. As a simple example, consider the following specification:

[2] A Prolog-like 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.

```
reaction(time(TimeAlarm),( out_r(alarm(TimeAlarm)) )).
```

When the tuple-centre current time reaches the `TimeAlarm` value (which must be instantiated to some numeric value), the reaction can be triggered, and its subsequent execution causes the insertion of the tuple `alarm` in the tuple centre.

Given that a timed reaction is conceptually triggered when the tuple-centre current time exactly matches the time specification in its head, each timed reaction is executed at most *once*: correspondingly, any triggered timed-reaction is automatically consumed after its execution, and so removed from the specification. Then, timed-reaction execution follows the same atomic semantics of normal reactions [11].

In ReSpecT tuple centres, it is possible to add and remove time reactions dynamically by exploiting the self-modifying specification predicates defined in ReSpecT: `out_r_spec`, `in_r_spec` and `rd_r_spec` predicates, which are used in to add, remove, and read reactions in general. In particular, the effect of an `out_r_spec(H,B)` is to add a reaction `reaction(H,B)` to the current specification, while `in_r_spec(H,B)` / `rd_r_spec(H,B)` removes / reads a reaction whose head and body match with `H` and `B`, respectively. This makes it possible to add and remove also timed-reaction specifications dynamically, by need.

For instance, the following specification

```
(1) reaction(out(clockStart), ( in_r(clockStart),
      current_time(StartTime), out_r(tick(StartTime)) ) ).

(2) reaction(out_r(tick(ClockTime)),(
      in_r(tick(ClockTime)), rd_r(delta_time(DeltaTime)),
      NewClockTime is ClockTime + DeltaTime,
      // any activity to be done at each clock goes here
      out_r_spec( time(NewClockTime), out_r(tick(NewClockTime)) ))).

(3) reaction(out(clockStop), ( in_r(clockStop),
      in_r_spec( time(ClockTime), out_r(tick(ClockTime)) ))).
```

defines a clock that starts when the tuple `clockStart` is first inserted in the timed tuple centre (reaction 1), cycles every `delta_time(@DeltaTime)` milliseconds (reaction 2), and stops when the tuple `clockStop` is finally inserted in the tuple centre (reaction 3).

3 An Example: Dining with Time Constraints

As a main example to show the effectiveness of the approach, we consider here an extension to the well-known *dining philosopher* problem, tackling the time issue. The dining philosopher is a classical problem used for evaluating the expressiveness of coordination languages in the context of concurrent systems [12]. In this problem, a number of philosophers eat at the same round table, using shared chopsticks. Philosophers alternate thinking with eating: two chopsticks are required to eat, no chopstick to think. Each philosopher shares the two chopsticks

on his left and right sides, respectively with the philosophers on his left and right sides. Coordination here is mostly needed to avoid deadlock, which can happen if each philosopher has taken a chopstick and is waiting for the other one, which is in turn taken by another waiting philosopher. In spite of its almost trivial formulation, the dining philosophers problem is generally used as an archetype for non-trivial resource-access policies.

The solution to the problem via ReSpecT consists in using a tuple centre — we call it `table` — for encapsulating the coordination policy required to decouple agent requests from the actual requests of resources — specifically, to encapsulate the management of chopsticks (for details refer to [7]). From the agent viewpoint, each philosopher *(i)* gets the two chopsticks needed by retrieving a tuple `chops(C1,C2)`, *(ii)* eats for a certain amount of time, *(iii)* provides back the chopsticks by inserting the tuple `chops(C1,C2)` in the tuple centre, and *(iv)* finally starts thinking until the next dining cycle. A process-algebraic-like description of this interactive behaviour is the following:

```
PHILO(C1,C2)::=
  THINK.table?in(chops(C1,C2)).EAT.table?out(chops(C1,C2)).PHILO(C1,C2)
```

The main point here is that philosophers do not need to worry about how to coordinate themselves, or how the resources are represented: they simply need to know which specific chopstick pair to ask for, and then they can focus on their main tasks (thinking and eating).

The tuple centre `table` is used as a coordination artifact to help their collective activity. Chopsticks are represented by `chop(`*`N`*`)` tuples, with *N* between 1 and the number of philosophers. Philosophers directly deal with couples of chopsticks (`chops/2` tuples). The tuple centre is programmed with the ReSpecT specification shown in Table 1 (top). Generally speaking, the coordinating behaviour accounts for mediating the representation of the resources (`chops/2` vs. `chop/1` tuples), and most importantly for avoiding deadlocks among the agents. In particular, if a philosopher requests a couple of available chopsticks, the request is reified by inserting a tuple `required` in the tuple centre (reaction 1). As this tuple is inserted, and if both the chopsticks are available, they are removed and a `chops` tuple is released to the agent (reaction 2). When the agent request is satisfied, the tuple `required` representing the pending agent request is removed (reaction 3). Then, when a philosopher inserts back the tuple representing the couple of chopsticks, the artifact reacts in order to mediate between the different chopsticks representations, by removing the `chops(`*`C1,C2`*`)` tuple (reaction 5) and inserting two separated chopsticks `chop(`*`C1`*`)` and `chop(`*`C2`*`)` (reaction 4). As a single chopstick is inserted, a control is made to check if such a chopstick is required by a pending agent request (`required` tuple) and — at the same time — if the other chopstick that appears in the pending agent request is available (reactions 6 and 7). In case, both chopsticks are removed, and the pending agent request is satisfied by producing a suitable `chops` tuple.

The basic formulation of the dining philosopher problem focuses on the deadlock issue. However, another relevant aspect for a correct collective behaviour of a MAS is fairness in the use of resources: once acquired the chopsticks, philosophers are meant to release them back, sooner or later. If a philosopher inciden-

Table 1. Timed ReSpecT specification for coordinating dining philosophers: (*top*) without maximum eating time constraints, (*bottom*) adaptation / extension to deal with timing constraints. In particular, reaction 8 is added to the specification, and reaction 4 is replaced by a new version (reaction 4')

```
1  reaction(in(chops(C1,C2)), (
     pre, out_r(required(C1,C2)))).

2  reaction(out_r(required(C1,C2)),(
     in_r(chop(C1)),in_r(chop(C2)),out_r(chops(C1,C2)))).

3  reaction(in(chops(C1,C2)), (
     post, in_r(required(C1,C2)))).

4  reaction(out(chops(C1,C2)), (
     out_r(chop(C1)),out_r(chop(C2)))).

5  reaction(out(chops(C1,C2)), (
     in_r(chops(C1,C2)))).

6  reaction(out_r(chop(C1)), (
    rd_r(required(C1,C)),in_r(chop(C1)),in_r(chop(C)),out_r(chops(C1,C)))).

7  reaction(out_r(chop(C2)), (
   rd_r(required(C,C2)), in_ r(chop(C)),inr(chop(C2)),out_r(chops(C,C2)))).
```

```
4' reaction(out(chops(C1,C2)), (
     in_r(used(C1,C2,T)),
     out_r(chop(C1)),out_r(chop(C2)))).

8  reaction(in(chops(C1,C2)), (
     post, current_time(T), rd_r(max_eating_time(Max)), T1 is T + Max,
     out_r(used(C1,C2,T)),
     out_r_spec(time(T1),(
          in_r(used(C1,C2,T)), out_r(chop(C1)),out_r(chop(C2)))))).
```

tally dies or refuses to release back the chopsticks, some philosophers can die by starvation and the coordination activity is compromised. So, a further issue that the coordination policy should capture is how to impose a constraint over the maximum time which philosophers can take to eat (i.e., to use the resources). Whenever such a constraint is violated, chopsticks released to philosophers are considered no more valid (no more usable), and new valid copies are re-created in the tuple centre, so as to allow the other philosophers to use them. If the

agent autonomy is to be preserved, such a coordinating behaviour should be obtained without forcing any individual agent behaviour: instead, this should be achieved by instrumenting the coordination artifact with suitable time-aware coordination laws.

Such a behaviour can straightforwardly be implemented with the ReSpecT model extended with time. By exhibiting the typical incremental nature of ReSpecT specifications, the previous (non-timed) specification is almost entirely reused, with only one minor change and one extension (reactions 4' and 8, respectively), as described in the bottom part of Table 1: the logic of the previous solution is mostly kept, and only the time-related aspects are specifically addressed by the two new reactions. Precisely, the behaviour is obtained by installing a timed reaction implementing a timeout every time a couple of chopsticks is retrieved by a philosopher (reaction 8), and keeping track of the chopstick currently in use by means of tuple `used`. The timed reaction is triggered after *Max* time units, where *Max* is the maximum eating time, stored in the `max_eating_time` tuple. When a philosopher inserts back the couple of chopsticks on time, the `used` tuple is successfully removed, along with the corresponding timed reaction, and the individual chopsticks are inserted back as in the non-timed case (compare reaction 4' with 4). If an installed timed reaction is triggered, it means that eating time for a philosopher has expired: then, tuple `used` is removed and the corresponding individual chopsticks (`chop` tuples) are re-created. When (if) a philosopher inserts back the couple of chopstick out of time, the related `used` tuple is no longer found, and the individual chopsticks are not inserted back (reaction 4' fails).

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 run time) according to the workload and, more generally, to environmental changes affecting the system.

4 Other Examples

In this section we describe how the extended model can be used to realise some other well-known coordination patterns based on the notion of time, namely timed requests and tuple leasing.

It is worth noting, however, that our point here is not to show that timed requests, or tuple leasing, can be expressed better by Timed ReSpecT than by the specific timed-primitives provided by JavaSpaces [10] and TSpaces [13]: those, in fact, are not general-purpose approaches, and can then address only a limited range of time-related coordination problems. Instead, the most relevant point here is the *generality* of our approach: here, in fact, the same simple model is shown to be capable to express time-based coordination policies of different kinds.

So, even the simple dining-philosophers problem extended with time constraints discussed in Section 3 can not be solved (at least, not straightfor-

wardly) with the timed primitives of JavaSpaces and TSpaces. Instead, the Timed ReSpecT model discussed above proves to be general enough to easily express timed requests and tuple leasing, as well as the timed dining-philosophers example.

4.1 Timed Requests

In this first example we model a timed `in` primitive, i.e. an `in` request that blocks only for an a-priori limited amount of time. An agent issues a timed `in` by

Table 2. Timed requests modelled using Timed ReSpecT

```
1  reaction(in(timed(MaxTime,Tuple,_)),(
     pre, out_r(required(MaxTime,Tuple)) )).

2  reaction(out_r(required(MaxTime,Tuple)),(
     in_r(Tuple),
     in_r(required(MaxTime,Tuple)),out_r(timed(MaxTime,Tuple,yes))
     ;
     current_time(Time), Timeout is MaxTime + Time,
     out_r_spec(time(Timeout),(
         in_r(required(MaxTime,Tuple)),
         out_r(timed(MaxTime,Tuple,no)))) )).

3  reaction(out(Tuple),(
     in_r(required(MaxTime,Tuple)),out_r(timed(MaxTime,Tuple,yes)))).
```

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 returned to the agent via unification with *`Template`*, with *`Res`* bound to the `yes` atom. Conversely, if no matching tuples are inserted within the specified *`Time`*, the request is unblocked by producing a suitable `timed` tuple with *`Template`* untouched and *`Res`* bound to `no`, which is then returned to the agent. Table 2 reports the Timed ReSpecT specification that implements the behaviour of this new primitive. When a timed `in` operation is issued, the request is reified by inserting a `required` tuple in the tuple centre (reaction 1). If a tuple matching the request is found, then the agent request is immediately satisfied by inserting back a `timed` tuple reporting a successful result (first part of reaction 2). Conversely, if no tuples are found, a timed reaction is installed, to be triggered after the amount of time specified by the agent request (second part of reaction 2)[3]. If a tuple matching the request is inserted on time (reaction 3),

[3] The `;` operator in ReSpecT has a meaning similar to the Prolog one: `(G1;G2)` succeeds if either `G1` or `G2` succeeds. More precisely, first `G1` is executed: if it fails, `G2` is then executed.

a `timed` tuple reporting a successful result is generated. Otherwise, if the timed reaction is triggered with the request still pending (tuple `required` is still in the tuple centre), a `timed` tuple reporting a negative result is generated, unblocking the agent request.

4.2 Tuples in Leasing

Finally, in this last example we model the notion of *lease*, analogously to the lease notion in models such as JavaSpaces [10] and TSpaces [13]. 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. The ReSpecT code implementing a simple form of leasing is:

```
reaction( out(leased(Tuple,LeaseTime)),(
   out_r(Tuple),
   current_time(Time), ExpireTime is Time + LeaseTime,
   out_r_spec( time(ExpireTime), in_r(Tuple) ))).
```

An agent inserts a tuple with a lease time by issuing an

```
out(leased(@Time,@Tuple))
```

According to the reaction described above, when a tuple with a lease time is inserted in the tuple centre, a timed reaction is inserted to be triggered when the leasing time has expired. The timed reaction simply removes the tuple in leasing. If the tuple is not found anymore (because it has been removed by some other agent request), the timed-reaction execution has no effect, for it simply fails.

5 Implementation Overview

The basic ReSpecT virtual machine has been designed and realised as a finite state automaton, with transitions through the basic stages (listening, speaking, reacting) as defined in the operational semantics described in [7, 11]. The tuple set, the pending query set, the triggered-reaction set and input/output event queues defined in [7] constitute the main data structures of the virtual machine. A Prolog engine is used for reaction triggering and execution; in particular ReSpecT primitives are implemented as Prolog built-in predicates defined in a library extending the basic engine. The technology is fully Java-based, and has been developed exploiting tuProlog, a Java-based Prolog engine, which is available as an open-source project at the tuProlog web site [14].

In the time-extended model, some new data structures are added:

- a clock, realised as a long-integer counter, holding current tuple-centre time expressed in milliseconds;
- a timed-reaction specification list, which is the list of timed reactions currently defined in the specification. The list is ordered by the time specified in the heads. The list is updated by the `out_r_spec` and `in_r_spec`, when inserting and removing timed reactions, and by the `set_spec` coordination primitive, when setting a specification which includes also time reactions;

– a timer service, triggering a tuple centre virtual machine in idle state at specific time points.

The Prolog library defining ReSpecT predicates has been extended with new predicates implementing the behaviour of the new temporal primitives (predicates).

The basic virtual-machine working-cycle has been extended so as to implement the time transition: at each cycle (that is, after any listening, speaking and reacting transition), current time is updated and the head timed-reaction specification list is checked: if there are timed-reaction specifications whose reaction time is less or equals than current tuple-centre time, they are removed from the list, and their bodies are added to the triggered-reaction set.

Also, to avoid problems due to idleness — when the timed-reaction list is not empty but no timed reaction have to be triggered yet, and there are no external requests to be served, no pending satisfiable pending requests, no triggered reactions to execute — the tuple-centre virtual machine properly configures the timer service, before going idle. In particular, the timer service is programmed so as to trigger the machine at the time point specified by the reaction time of the first timed reaction of the list.

6 Related Works

Outside the specific context of coordination models and languages, the issue of defining suitable languages for specifying the communication and coordination in timed systems have been extensively studied. Examples of such languages are Esterel [15] and LUSTRE [16], both modelling synchronous systems, the former with an imperative style, and the latter based on dataflow. In the coordination literature several approaches have been proposed for extending basic coordination languages with timing capabilities. [17] introduces two notions of time for LINDA-style coordination models, relative time and absolute time, providing for a number of time-related features. Time-outs have been introduced in JavaSpaces [10] and in TSpaces [13], and have been generally formalised by Timed Linda [18].

The Timed ReSpecT approach described in this work differs from these approaches for at least two main reasons. First of all, Timed tuple centres extend the tuple-centre model without altering the basic LINDA model: LINDA primitives are kept unchanged (no change to their semantics, no timed primitives added), and the extension focuses instead on the expressiveness and behaviour of the coordination medium. Also, Timed ReSpecT does not provide agents with specific time capabilities, but — following the philosophy of programmable coordination media [19] — aims instead at instrumenting the model with the general expressiveness required to capture any time-based coordination pattern.

7 Conclusions

The first attempt to enhance ReSpecT with time is reported in [20]. There, however, the syntax and the semantics of the extension (based on the notion

of trap) are quite *ad hoc*, and do not fit well the original ReSpecT model: the examples reported there can be easily compared with the ones in this article to clearly appreciate the differences between the two approaches. Moreover, the contribution provided by this work is meant to be broader, since it generalises over the notion of tuple centre, and extends to the design and development of general-purpose time-aware coordination artifacts in MASs [1].

Our approach aims to be general and expressive enough to allow for the description of a wide range of coordination patterns based on the notion of time, by exploiting medium programmability and the basic time-based mechanisms. An important feature exhibited by our approach is the *encapsulation of coordination*: embedding (specifying, enacting) time-aware policy directly inside the coordination medium promotes modularity of the coordination programs, and then reusability and extensibility. Temporal features have been added with no changes to the usage interface of tuple centres, which is still based on the basic set of Linda-like coordination primitives. Also, the extension has been realised while preserving all the essential properties of the ReSpecT model: in particular, reaction execution is still atomic (both at the system and at the agent levels [7]), and reactions are executed sequentially. Even more, the declarative nature of the reactions, along with the execution model, makes (Timed-)ReSpecT mostly incremental in its specifications, as shown by the example discussed in Section 3.

In the implementation of the model, the issue of the centralised vs. distributed implementation of tuple centres arises. The basic tuple centre model does not necessarily require a centralised implementation *per se*: however, the extension provided in this work — devising out a notion of time for each individual medium — leads quite inevitably to realise tuple centres with a specific spatial location. This is what already happens in the TuCSoN coordination infrastructure, where there can be multiple tuple centres spread over the network, collected and localised in infrastructure nodes. It is worth mentioning that this problem is not caused by our framework, but is somehow inherent in 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 for a distributed implementation of an individual coordination medium is an issue of some relevance only for very specific application domains. For most 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 MAS 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/she is going to use, and the quality of service (robustness in particular) ensured by the infrastructure.

Even though the model of time used here is not meant to deal with real-time issues in any way, we understand that this work could provide us with a solid grounding for soft and hard real-time agent coordination. In the future, we mean to explore real-time issues by suitably extending (for instance, with time-labelled triggered reactions) the time model presented here.

References

1. Omicini, A., Ricci, A., Viroli, M., Castelfranchi, C., Tummolini, L.: Coordination artifacts: Environment-based coordination for intelligent agents. In Jennings, N.R., Sierra, C., Sonenberg, L., Tambe, M., eds.: 3rd international Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004). Volume 1., New York, USA, ACM (2004) 286–293
2. Viroli, M., Ricci, A.: Instructions-based semantics of agent mediated interaction. In Jennings, N.R., Sierra, C., Sonenberg, L., Tambe, M., eds.: 3rd international Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004). Volume 1., New York, USA, ACM (2004) 102–110
3. Viroli, M., Omicini, A.: Coordination as a service: Ontological and formal foundation. Electronic Notes in Theoretical Computer Science **68** (2003) 1st International Workshop "Foundations of Coordination Languages and Software Architecture" (FOCLASA 2002), Brno, Czech Republic, 24 August 2002. Proceedings.
4. Omicini, A., Zambonelli, F.: Coordination for Internet application development. Autonomous Agents and Multi-Agent Systems **2** (1999) 251–269 Special Issue: Coordination Mechanisms for Web Agents.
5. TuCSoN: Home page. `http://lia.deis.unibo.it/research/TuCSoN/` (2001)
6. Gelernter, D.: Generative communication in Linda. ACM Transactions on Programming Languages and Systems (TOPLAS) **7** (1985) 80–112
7. Omicini, A., Denti, E.: From tuple spaces to tuple centres. Science of Computer Programming **41** (2001) 277–294
8. Denti, E., Natali, A., Omicini, A.: On the expressive power of a language for programming coordination media. In: 1998 ACM Symposium on Applied Computing (SAC'98), Atlanta, GA, USA, ACM (1998) 169–177 Special Track on Coordination Models, Languages and Applications.
9. Ricci, A., Omicini, A., Viroli, M.: Extending ReSpecT for multiple coordination flows. In Arabnia, H.R., ed.: International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'02). Volume III., Las Vegas, NV, USA, CSREA Press (2002) 1407–1413
10. Freeman, E., Hupfer, S., Arnold, K.: JavaSpaces: Principles, Patterns, and Practice. The Jini Technology Series. Addison-Wesley (1999)
11. Omicini, A., Denti, E.: Formal ReSpecT. Electronic Notes in Theoretical Computer Science **48** (2001) 179–196 Declarative Programming – Selected Papers from AGP 2000, La Habana, Cuba, 4–6 December 2000.
12. Dijkstra, E.: Co-operating Sequential Processes. Academic Press, London (1965)
13. Wyckoff, P., McLaughry, S.W., Lehman, T.J., Ford, D.A.: T Spaces. IBM Journal of Research and Development **37** (1998) 454–474
14. tuProlog: Home page. `http://lia.deis.unibo.it/research/tuProlog/` (2001)
15. Berry, G., Gonthier, G.: The Esterel synchronous programming language: Design, semantics, implementation. Science of Computer Programming **19** (1992) 87–152
16. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.A.: LUSTRE: a declarative language for real-time programming. In: 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, ACM Press (1987) 178–188
17. Jacquet, J.M., De Bosschere, K., Brogi, A.: On timed coordination languages. In Porto, A., Roman, G.C., eds.: Coordination Languages and Models. Volume 1906 of LNCS., Springer-Verlag (2000) 81–98 4th International Conference (COORDINATION 2000), Limassol, Cyprus, 11–13 September 2000. Proceedings.
18. de Boer, F., Gabbrielli, M., Meo, M.C.: A Timed Linda language and its denotational semantics. Fundamenta Informaticae **63** (2004) 309–330

19. Denti, E., Natali, A., Omicini, A.: Programmable coordination media. In Garlan, D., Le Métayer, D., eds.: Coordination Languages and Models. Volume 1282 of LNCS., Springer-Verlag (1997) 274–288 2nd International Conference (COORDINATION'97), Berlin, Germany, 1–3 September 1997. Proceedings.
20. Ricci, A., Viroli, M.: A timed extension of ReSpecT. In: 2005 ACM Symposium on Applied Computing (SAC 2005), Santa Fe, NM, USA, ACM (2005) 420–427 Special Track on Coordination Models, Languages and Applications.