A Timed Extension of ReSpecT

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

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

ABSTRACT

Among the existing extensions to the basic tuple space coordination model, the tuple centre approach has been introduced to allow for the flexible programming of a tuple space behaviour, so as to encapsulate coordination laws directly as behaviour of the coordination medium. In particular, the logic-based language ReSpecT has been used for programming tuple centres in the TuCSoN coordination infrastructure [11, 9].

However, among the application contexts that can be suitably engineered with agents and coordination infrastructures, some involve coordination processes where the notion of time and duration play a relevant role. Examples include distributed control systems, protocol-based interactions as in auctions, and in general all the coordination contexts where high dynamism and openness are concerned, which call for time-aware coordination artifacts supporting timed system engineering.

Accordingly, in this work we discuss how the basic ReSpecT tuple centre model has been extended to support the definition and enactment of time-aware coordination policies. Several examples are provided to show the expressiveness of the language to model temporal coordination primitives and laws.

Categories and Subject Descriptors

D.1.3 [Programming Techniques]: Concurrent Programming; D.2.11 [Software Engineering]: Software Architectures; D.3.3 [Programming Languages]: Language Constructs and Features

Keywords

Coordination Models, Timed Systems, Tuple Centres

1. INTRODUCTION

Since its invention, the tuple space coordination model [5] has been extended in several ways [12]. Several approaches extended the basic model adding new coordination primitives such as collect or rdall [13]; others adopted different kind of communication languages, ranging from logic tuples, to objects and XML documents [14]. The tuple centre approach [9] extends the basic model by making the coordination medium programmable, so as to encapsulate coordination laws directly as media behaviour. ReSpecT in particular has been used as logic-based language for programming tuple centre behaviour [8]. This extension can be considered a generalisation of the previous ones: it makes it possible for instance to define new coordination primitives or realising event based coordination patterns by suitably programming the coordinating behaviour of tuple centres [3]. ReSpecT tuple centres are the core of agent coordination infrastructures such as TuCSoN [11].

In ReSpecT a coordination algorithm is defined basically specifying how the tuple set must change when a communication event occurs as the result of an interaction between an agent and the tuple centre. Since ReSpecT is Turing equivalent [9], any algorithm can be encoded, and then any coordination specification can be defined. This characterisation however does not include time-based algorithms, i.e. the capability of specifying coordination patterns and strategies based in some way on the notion of time. In fact, several examples of coordination primitives and laws exist that involve the notion of time. 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 be the basis for structuring more complex coordination scenarios, such as e.g. auctions and negotiations protocols including time-based guarantees and constraints. In general, time plays a decisive role in several coordination contexts, in particular in distributed control systems, service orchestrations, etc.

In this work we discuss how the basic ReSpecT tuple centre model has been extended to support the definition and enactment of coordination policies for timed systems. In this context, this work can be considered useful for the design and development of time-aware coordination artifacts [10] for agent based engineering of timed contexts. 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
medium. From the discussion above, it is clear that the extension must be expressive enough to capture temporal coordination primitives of existing models and systems.

Also, the extension can be useful to augment the robustness of the basic ReSpecT model with respect to unwanted events or bugs in the coordination specification, which is indeed a relevant issue for the engineering of open multi-agent systems [7]. In our case, a bug in a ReSpecT program can result in a not terminating behaviour of a tuple centre when reacting to a communication event, caused by infinite chains of reactions: the extension developed here then could be exploited to generate traps in order to eventually break such chains.

The rest of the paper is organised as follows: Section 2 discusses in details the ReSpecT extended model, Section 3 provides some concrete examples exploiting the extended model, Section 4 describes some aspects of the implementation, Section 5 provides some reflections on the features of the approach and finally Section 6 provides related works, conclusion and future works.

2. 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).

2.1 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 \( \text{in}(t) \)), removing a tuple matching template \( tt \) (by primitive \( \text{in}(tt) \)), and reading a tuple matching template \( tt \) (by primitive \( \text{rd}(tt) \))\(^1\). 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 re-actively 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 \( \text{in}, \text{rd}, \text{out} \)), or (ii) a speaking, the production of a reply towards a coordinated process (either the reply to a \( \text{in} \) or \( \text{rd} \))\(^2\).

The ReSpecT language can be used to declare a set \( \sigma \) 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

\[ \sigma := \{ \text{reaction}(p(t), \text{body}) \} \]

\[ p := \text{cp} \mid \text{rp} \]

ReSpecT primitives

\[ \text{cp} := \text{out} \mid \text{in} \mid \text{rd} \]

Comm. primitives

\[ \text{rp} := \text{in} \mid \text{rd} \mid \text{out} \mid \text{no} \]

Reaction primitives

\[ \text{body} := [\text{goal}, \text{goal}] \]

Specification body

\[ \text{ph} := \text{pre} \mid \text{post} \]

Direction predicates

\[ \text{goal} := \text{ph} \mid \text{rp}(t) \]

Goals

Figure 1: The syntax of a ReSpecT specification

a body, reactions are composed by a sequence of reaction primitives \( \text{rp} \) resembling LINDA primitives, which are used to remove a tuple (\( \text{in} \)), read a tuple (\( \text{rd} \)), insert a tuple (\( \text{out} \)), and check for the absence of a tuple (\( \text{no} \)). This sequence can contain a direction predicate \( \text{ph} \mid \text{pre} \) or \( \text{ph} \mid \text{post} \), which is used to filter between reactions to a listening or a speaking. In particular, here we consider therefore five kinds of external communication events: listening of a \( \text{out}, \text{rd}, \text{or in} \), and speaking of a \( \text{in} \) and \( \text{rd} \).

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 \( \text{in}, \text{rd}, \text{and no} \) 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.

2.2 The Extended Model
The complete syntax of the extended model is given in Figure 2. 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:

\[ \text{current.time}(?Tc) \]

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

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

3In current implementation the temporal unity is the millisecond

4A 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.
This primitive (predicate) is successful if \( T_c \) (typically a variable) unifies with the current tuple centre time \( T_c \). As an example, the reaction specification tuple

\[
\text{reaction}(\text{in}(p(X)),\langle \text{current_time}(T_c), \text{out}_r(\text{request_log}(T_c,p(X))) \rangle)\]

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 of a (trap) source, characterised by a unique identifier \( ID \), a time \( T_e \) and a description tuple \( T_d \). 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 \( \text{expired}(T) \) a certain interval \( T_e \) after the insertion of a tuple \( \text{leased}(T) \);

- 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 \( \text{expired}(T) \) occurs.

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

- \( \text{new_trap}(-ID,0,T_e,+T_d) \)
- \( \text{kill_trap}(0,ID) \)

The first is successful if \( T_e \) 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 \( T_c \) will be equal or greater than current time plus \( T_e \), a trap event described by the tuple \( T_d \) 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 \( \text{ReSpecT} \) semantics, if the reaction including an invocation to primitive \ new_trap \ fails, no trap source is actually installed. An example involving the \ new_trap \ primitive is as follows:

\[
\text{reaction}(\text{out}(\text{leased}(T_c,T_e)),\langle \text{new_trap}(-,\text{leased}(T_c,T_e),\text{expired}(T)) \rangle)\]

The reaction is triggered when a tuple matching \( \text{leased}(T_c,T_e) \) is inserted, and it installs a new trap source which will generate a trap described by the tuple \( \text{expired}(T) \) after \( T_e \) units from then. Primitive \ kill_trap \ is instead used to deinstall a source given its identifier: such a primitive fails if no installed sources are characterised by the provided identifier.

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:

\[
\sigma ::= \{ \text{reaction}(\text{in}(p(t)),(\text{body})), \}
\]

<table>
<thead>
<tr>
<th>Reaction primitives</th>
<th>Comm. primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{p} ::= \text{cp} \mid \text{rp} \mid \text{tr} )</td>
<td>( \text{cp} ::= \text{out} \mid \text{in} \mid \text{rd} )</td>
</tr>
<tr>
<td>( \text{cp} ::= \text{in} \mid \text{rd} )</td>
<td>( \text{rp} ::= \text{in}_r \mid \text{rd}_r )</td>
</tr>
<tr>
<td>( \text{out}_r \mid \text{no}_r )</td>
<td>( \text{body} ::= \text{goal}(</td>
</tr>
<tr>
<td>( \text{ph} ::= \text{pre} \mid \text{post} )</td>
<td>( \text{goal} ::= \text{ph} \mid \text{rp}(t) \mid \text{tp} )</td>
</tr>
<tr>
<td>( \text{tr} ::= \text{trap}(t) )</td>
<td>( \text{tp} ::= \text{new}<em>\text{trap}(\text{id},\text{time},t) \mid \text{kill}</em>\text{trap}(\text{id}) \mid \text{current_time}(t) )</td>
</tr>
</tbody>
</table>

Figure 2: The syntax of an extended \( \text{ReSpecT} \) specification

\( \text{reaction}(\text{trap}(\text{Tuple})), \text{Body} \)

Body specifies the set of actions to be executed when a trap with a description tuple matching the template \( \text{Tuple} \) occurs. In the following simple example

\[
\text{reaction}(\text{trap}(\text{expired}(T)),(\text{in}_r(T)))\]

when a trap described by a tuple matching the template \( \text{expired}(T) \) occurs, the tuple specified in \( T \) is removed from the tuple set. Notice that if the tuple is not present the \( \text{in}_r \) 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 \( \text{ReSpecT} \) 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 priority of reactions fired by trap events over standard reactions. 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 \( \text{ReSpecT} \) model: traps event in the trap queue are listened (and related reactions executed) after the completion of any reaction possibly in execution. Then, chains of reactions can be broken, not individual reactions. This is fundamental in order to preserve the semantic properties of \( \text{ReSpecT} \) 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 5, the possibility of breaking reaction chains is important to build robust coordinating behaviour, in particular with respect to possible bugs generating terminating reaction chains.
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 is worth noting that most of these examples appear in previous literature as core primitives for timed extensions of coordination models: our general approach allows to support them all on top of the same model.

3.1 Timed Requests

In this first example we model a timed \texttt{in} primitive, i.e. an \texttt{in} request that keeps blocked only for a maximum amount of time. An agent issues a timed \texttt{in} by executing primitive \texttt{in timed(@Time, ?Template, -Res)}. If a tuple matching \texttt{Template} is inserted within \texttt{Time} units of time, the requested tuple is removed and taken by the agent as usual with \texttt{Res} being bound to the \texttt{yes} atom. Conversely, if no matching tuples are inserted within the specified time, \texttt{Res} is bound to \texttt{no} atom. Table 1 reports the ReSpecT specification which makes it possible to realise the behaviour of this new primitive. When the \texttt{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 \texttt{trap info} is inserted in the tuple set, reifying information about the installed trap

Table 1: ReSpecT specification for modelling a timed in primitive

Nevertheless, it is worth mentioning here that other semantics are possible and interesting. By giving highest priority to the standard execution, one would ensure that traps never interfere with it. In exchange of the better isolation of code achieved, however, the same timing constraints can no longer be guaranteed: 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 external communication events) at the time its source is installed\(^5\).

3.2 Tuples in Leasing

In this third example we model the notion of \texttt{lease}, analogously to the lease notion in models such as JavaSpaces \cite{4} and TSpaces \cite{17}. Tuples can be inserted in the tuple set specifying a lease time, i.e. the maximum amount of time for which they can reside in the tuple centre before automatic removal.

An agent inserts a tuple with a lease time by issuing an \texttt{outleased(@Time, @Tuple)}. Table 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 \texttt{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 \texttt{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 \texttt{rd} has been satisfied (reaction 3). An \texttt{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 \texttt{out1} tuple carrying information about the presence of the leased tuple is removed (reaction 5).

3.3 Lasting Removal

In this third example a temporal extension of the well-known \texttt{inall} primitive is described. The \texttt{inall} primitive has been introduced in literature to extend the basic LINDA model with the capability of remove all the tuples matching a template atomically. Here we extend this primitive by

Table 2: ReSpecT specification for modelling tuples with a lease time

\(^5\)This interesting feature which is subject of current research is not described in this paper for brevity.
The coordination specification in ReSpecT (first 6 reactions of Table 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. If one such deadline expires the chopsticks are regenerated in the tuple centre, so as to avoid the starvation of the philosophers waiting for them, and the chopsticks eventually inserted out of time are subsequently 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 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 its 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.

### 4. 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 operation semantics described in [9, 8]. The tuple set, the pending query set, the triggered reaction set and input/output event queues defined in [9] constitute the main data structures of the virtual machine. A Prolog engine is used for reaction triggering and execution; in particular ReSpecT primitives are realised 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 (available as open source project at the tuProlog web site [16]).

In the extended model some new data structure are added:

- A trap event queue — Trap events are represented by a related trap event class; the queue is used to keep

---

**Table 3: ReSpecT specification mimicking an inall with a duration time**

giving it a duration time. An agent invokes the primitive by issuing an inall_timed(0/Time, +Tuple, !OutList): from then and during Time units of time, tuples matching the template Tuple are removed from the space and gathered in the OutList list provided when the primitive expires. As for the previous examples, a ReSpecT specification realising the desired behaviour is reported — see Table 3. Briefly, when the inall request is issued, a trap source is installed for generating the trap event when the specified amount of time is elapsed, and a reaction chain (involving reactions 2 and 3) is triggered for (i) removing all the tuples currently present in the tuple set matching the specified template and (ii) collecting and reifying them in a current_in_all tuple (reaction 1). Each tuple inserted in the tuple centre matching the template specified in a pending inall is inserted in the list stored in current_in_all tuple (reaction 4). Finally, when the trap event occurs, the current_in_all tuple is removed and an all_timed tuple matching the first in request is inserted, carrying the list of collected tuples (reaction 5).

#### 3.4 Dining Philosophers with Maximum Eat-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) 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 pseudo-code reflecting this interactive behaviour is the following:

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

The coordination specification in ReSpecT (first 6 reactions of Table 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. If one such deadline expires the chopsticks are regenerated in the tuple centre, so as to avoid the starvation of the philosophers waiting for them, and the chopsticks eventually inserted out of time are subsequently 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 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 its 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.
A trap timer — A timer service enabling the registration and execution of independent tasks — in our case instances of a trap timer task class — at specific temporal deadlines. The timer service is based directly on a Java class provided in the JDK. The trap timer task class extends the basic timer task class — also this one provided by the JDK —, specialising the actions to be executed when the deadline is reached, which consists in the creation of a trap event, its insertion in the trap event queue and the removal of the trap generator from the related set.

A trap generator set — a set used to keep track of trap generators dynamically instantiated by reactions. A trap generator class encapsulates the basic information characterising a trap generator — such as the tuple to be inserted in the tuple set when the trap occurs — and a reference to the trap timer task executed when the timeout expires.

A trap — A timer service enabling the registration and execution of independent tasks — in our case instances of a trap timer task class — at specific temporal deadlines. The timer service is based directly on a Java class provided in the JDK. The trap timer task class extends the basic timer task class — also this one provided by the JDK —, specialising the actions to be executed when the deadline is reached, which consists in the creation of a trap event, its insertion in the trap event queue and the removal of the trap generator from the related set.

The Prolog library defining ReSpecT predicates has been extended with new predicates implementing the behaviour of the new primitives. In particular, new_trap predicate creates a new trap generator, inserts it in the trap generator set and registers its trap timer task in the timer service of the virtual machine; kill_trap predicate removes an existing trap generator from the trap generator set, deregistering also its trap timer task from the timer service.

Finally, the transitions of the basic virtual machine have been extended in order to (i) listening trap events as soon as the trap event queue is not empty, which consists in collecting reactions triggered by them and adding these reactions in the triggered trap reaction set; (ii) executing one by one the trap reactions collected in the triggered trap reaction set, as soon as this set is not empty.

The technology is available with the version 1.4.0 of TuCSoN infrastructure, available (as open source project) at the TuCSoN web site [15].

5. 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 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 at certain time points. With respect to this approach and also to other approaches, the solution described in this work has several advantages:

- Encapsulation 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.

Table 4: ReSpecT specification for coordinating dining philosophers with a maximum eating time

<table>
<thead>
<tr>
<th>Reaction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reaction(in(chops(C1,C2)), (pre, out_r(required(C1,C2))))</td>
<td>% a request of the chopsticks is reified with a required tuple</td>
</tr>
<tr>
<td>reaction(out_r(required(C1,C2)), (in_r(chop(C1)), in_r(chop(C2)), out_r(chops(C1,C2))))</td>
<td>% if both the chopsticks are available, a chops tuple is generated</td>
</tr>
<tr>
<td>reaction(in(chops(C1,C2)), (post, in_r(required(C1,C2))))</td>
<td>% with the retrieval of the chops tuple, the chopsticks request % is removed</td>
</tr>
<tr>
<td>reaction(out(chops(C1,C2)), (current_agent(AgentId), no_r(chops(AgentId,C1,C2)), in_r(chop(C1,C2)), out_r(chop(C1,C2))))</td>
<td>% the release of a chops tuple on time causes the % insertion of individual chopsticks, represented by the two % chop tuples</td>
</tr>
<tr>
<td>reaction(out_r(chop(C1)), (rd_r(required(C1,C2)), in_r(chop(C1)), out_r(chops(C1,C2))))</td>
<td>% a chops tuple is generated if there is a pending request, % and both chop tuples are actually available</td>
</tr>
<tr>
<td>reaction(out(chop(C2)), (rd_r(required(C1,C2)), in_r(chop(C2)), out_r(chops(C1,C2))))</td>
<td>% a chops tuple requests causes also creating a new trap generator, % keeping track of its information in the chops_pending_trap tuple</td>
</tr>
<tr>
<td>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))))</td>
<td>% when chopsticks are released on time, the trap generator is removed</td>
</tr>
<tr>
<td>reaction(out(chops(C1,C2)), (in_r(chops_pending_trap(ID,C1,C2)), kill_trap(ID))))</td>
<td>% trap generation causes the insertion back of the missing tuples % and the insertion of a tuple keeping track of the invalid chopsticks</td>
</tr>
<tr>
<td>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))))</td>
<td>% chopsticks released that are invalid (due to time expiration) are % immediately removed</td>
</tr>
</tbody>
</table>
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 characterised by soft timing constraints. Even if not suitable for hard real time, 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 generated by a timer agent 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 highest priority. Also, given a reaction specification, it is possible to determine the maximum delay time which can elapse since the occurrence of a trap event and the execution of its related reactions.

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. This is not reported here for the sake of space, but will be presented in a forthcoming extended version of this paper.

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.

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 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 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 to have a distributed implementation of individual coordination media is a real issue 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 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 it is going to use, and the quality of service (robustness in particular) provided by the infrastructure.

6. 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 timed 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 the coordination literature several approaches have been proposed for extending basic coordination languages with timing capabilities. [6] 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 [17].

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.

Ongoing and future works mainly account for two aspects: providing a formal model coherent and compatible with the operational semantics defined in [9, 8], and assess the approach with more complex and real-world application scenarios.

7. REFERENCES


