Artifacts for Time-Aware Agents *

Cédric Dinont
USTL/LIFL and ISEN
59655 Villeneuve-d'Ascq, France
cedric.dinont@lifl.fr

Emmanuel Druon
ISEN 41 Bd Vauban
59046 Lille Cedex, France
emmanuel.druon@isen.fr

Philippe Mathieu
USTL/LIFL
59655 Villeneuve-d'Ascq, France
philippe.mathieu@lifl.fr

Patrick Taillibert
Thales Aerospace Division
78951 Elancourt, France
patrick.taillibert@fr.thalesgroup.com

ABSTRACT

Time-aware agents are agents capable of reasoning about their tasks duration and deadlines, and, more generally, to manage the temporal aspects of the execution of their tasks. We first focus on the case of agents in charge of long duration computations, sustaining that it is not acceptable for an autonomous agent to remain unaware of its environment for too long. We then consider deadline meetings when several time-aware agents share the same CPU. To achieve these goals, we recognize the importance of the artifact concept [16]. We introduce computational artifacts for long duration tasks and a coordination artifact for managing the CPU agenda and acting as an intermediary when agents negotiate CPU power. Control of computational artifacts is done thanks to a set of operating instructions dynamically computed by the coordination artifact.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Multiagent systems*

General Terms

Algorithms

Keywords

Coordination, CPU sharing, Artifacts

1. INTRODUCTION

The context of this paper is the design of intelligent agents that make use of algorithms stemming from Artificial Intelligence research. For example, planification problems in cognitive agents make use of complex algorithms, often heuristic in nature.

Using such algorithms with very high complexity leads to execution times highly variable. A problem search space can have critical regions: for a small variation of input data, the computing time necessary to find the solution can tremendously vary and even become prohibitive. Processor resources needs are thus very fluctuating and difficult to anticipate. That concerns a lot of problems which are solved by using more or less powerful heuristics.

The implementation of such systems should take into account the following difficulties:

(i) *agents may "lose consciousness".* Even if involved in long duration tasks, agents should stay permanently aware of their environment and should be able to process information sent by others. On the contrary, an agent that have to execute long duration algorithms would be deconnected from the outside world and would not even realize that its calculation has become useless if a received but not processed message had said so. It would unnecessarily use CPU power that could be useful to other tasks or other agents.

(ii) *agents may be unable to meet their commitments.* The notion of commitment is essential to the rationality of agents in a multi-agent context. When time is concerned, an agent must be able to guarantee results within a given deadline, even if it means modifying its strategy, using an alternative algorithm or giving a solution of less quality, while respecting its deadlines.

(iii) *agents may share the same CPU.* This is most often the case that several agents of a MAS run on the same CPU and therefore have to share CPU power. In that case, it is very difficult to ensure that time constraints will be respected. Solution have to be found since it is generally unacceptable to assume that agents do not share the same CPU.

A natural solution to the previous difficulties would be to split agents having to achieve long duration computation in two or more parts:

(i) a "main agent", that would remain aware of its environment all the time and hence open to information coming from other agents,

*This work is partially financed by Thales Aerospace Division and the région Nord-Pas-de-Calais.

AAMAS'06 May 8–12 2006, Hakodate, Hokkaido, Japan.

(ii) one or more "computational agents" in charge of the long duration computations, the initial agent was concerned with.

It sounds like a good approach, but one may ask if these computational agents are really autonomous? Their purpose is to work on potentially long computations, so they can stay away from their environment for long periods of time. Worse, in order to manage the CPU sharing, the main agent would need some control facilities allowing it to suspend, restart or even completely stop the execution of its associated computational agents if necessary.

So even if this solution is technically the good one, one may legitimately be reluctant to call an agent an entity with such a damaged autonomy. The solution is to rely upon the notion of artifact proposed in [20, 17] in place of computational agents. An artifact is a first class entity of any MAS giving some kind of functionality that agents *use* to achieve their goals.

We will recall artifacts and their modeling in section 2.1 and explain in section 2.2 the way they are used for embodying our computational agents. We can already mention that even if it will solve the first difficulty previously stated (avoiding agent "lost of consciousness"), meeting deadline commitments and sharing CPU still remain an open problem. This is why we propose in section 3 to use a coordination artifact which will be in charge of determining time slices allocated to each computational artifact and providing the main agent with a set of operating instructions to control its artifacts. Section 4 explains the way the coordination artifact behaves. Section 5 gives results obtained on our implementation and section 6 discusses related and future works in the context of time management in MAS.

2. COMPUTATIONAL ARTIFACTS

We are interested in time-aware agents when they have to run one or several long duration tasks while being able to meet the deadlines they are committed to. It requires that the long duration tasks are run in processes separated from the one running the agent since, in the other case, the agent might lose control over its own behaviour, obliged to wait for the end of the tasks before taking any decision.

A straightforward (but costly) solution would be to run each task on a separate processor. Nonetheless, in most of the cases, CPU has to be shared and hence the agent must be able to control the tasks, that is to decide when to start, suspend, restart or cancel them in order to meet its own deadlines. Long duration tasks could be implemented as agents but one can legitimately wonder about the autonomy of an agent that can be externally controlled!

That every item of a multiagent system can be implemented as an agent is a point that has already been raised by Ricci, Viroli and Omicini [20, 17, 16], especially in the case of coordination agents. Their answer was to introduce *artifacts* as a first-class abstraction used to design and program those aspects of multiagent systems for which the agent abstraction is not suitable. We suggest that the same approach has to be adopted for the time-aware agents we are concerned with.

The following section 2.1 summarizes the *artifact* concept, directly borrowing from [20] the process algebra based formalism they introduced. Section 2.2 proposes the *computational artifact* as an abstraction for implementing the computational activities of time-aware agents.

2.1 The Notion of Artifact

As presented in [20], if agents are conceived as goal-governed or goal-oriented systems, an artifact is an entity an agent *uses* to achieve its goals. Artifacts provide a certain function that can be exploited as a service.

An artifact is characterized by:

- *a usage interface (UI):* defined in terms of a set of operations. These operations are of two kinds: action execution and perception about the completion of an action.
- *a set of operating instructions (OI):* the (formal) description of how agents can use the artifact.
- *a behavior specification:* the (formal) description of the internal behavior of the artifact.

Note that, taking the agent point of view, to exploit a coordination artifact means to follow its operating instructions, on a step-by-step basis. Thus, the agent does not have to know *how* the functionality is implemented.

We now briefly present the formal model introduced in [20] that makes use of a process algebra to express operating instructions. An instruction $I \in \mathcal{I}$ is expressed as:

$$I ::= 0 \mid !\alpha \mid \underline{\alpha} \mid ?\pi \mid I; I \mid I + I \mid (I||I) \mid \mathcal{D}(t_1, \ldots, t_n)$$

It is defined out from atomic instructions through structured instructions. Atomic instructions include the void behavior (0), the execution of an action α ($!\alpha$), an action α being started but not yet completed ($\underline{\alpha}$) and a perception π of action completion ($?\pi$). Structured instructions are the sequential composition of two instructions by operator ";", choice by "+", and parallel composition by "||". An instruction can also be an invocation $\mathcal{D}(t_1, \ldots, t_n)$ of antoher operating instruction. Several structural congruence rules that we do not mention here express commutativity and transitivity of previous operators.

Operating instructions evolve as the agent interacts with the artifact. The possible interaction acts $\delta \in \Delta$ are the execution of action α (α) and the perception π of the completion of α ($\alpha : \pi$):

$$\delta ::= \alpha \mid \alpha : \pi$$

This language can be given operational semantics. In the following rules, notation $I \xrightarrow{\delta}_{\mathcal{I}} I'$ means that OI state $I \in \mathcal{I}$ moves to new state $I' \in \mathcal{I}$ by the occurrence of interaction act $\delta \in \Delta$. Notation $[t/t']$ is used to indicate the substitution of t by t' in the continuation of instructions.

$I\|\|I'' \xrightarrow{\delta}_{\mathcal{I}} I'\|\|I''$	if $I \xrightarrow{\delta}_{\mathcal{I}} I'$	[PAR]
$I + I' \xrightarrow{\delta}_{\mathcal{I}} I''$	if $I \xrightarrow{\delta}_{\mathcal{I}} I''$ and $I' \not\xrightarrow{\delta}_{\mathcal{I}} I'''$	[ECH]
$I + I' \xrightarrow{\delta}_{\mathcal{I}} I'' + I'''$	if $I \xrightarrow{\delta}_{\mathcal{I}} I''$ and $I' \xrightarrow{\delta}_{\mathcal{I}} I'''$	[LCH]
$!\alpha; I \xrightarrow{\alpha'}_{\mathcal{I}} \underline{\alpha'}; [\alpha/\alpha']I$		[ACT]
$\underline{\alpha}; ((?\pi; I) + I'); I'' \xrightarrow{\alpha:\pi'}_{\mathcal{I}} [\pi/\pi'](I; I'')$		[COMP]

Rule [PAR] specifies that only one OI is followed at a given time. Rule [ECH] defines the semantics of exclusive choice: if I is executed, choices inside I' are excluded. Rule [LCH]

instead defines late choice: if interaction δ allows multiple choices, δ is executed and the choice is deferred to a forthcoming interaction. Rule [ACT] specifies that if α is the next action to execute, a specialization α' of α can be executed, propagating the corresponding substitution in the continuation of instructions. Rule [COMP] similarly deals with perception of the completion of an action.

Viroli and Ricci proposed in [20] an extension to the previous algebra to take into account the notion of timeout in OIs. The definitions of instructions and interaction acts are extended as follows:

$$I ::= \ldots \mid T(n) \mid \epsilon \qquad \delta ::= \ldots \mid \tau(m)$$

$T(n)$ is the timeout instruction, with parameter n being the duration of the timeout. ϵ is an error state the OI moves to when a timeout has expired while the OI was waiting for an interaction. $\tau(m)$ is the perception of m clock ticks. As for previous instructions, following rules give an operational semantics to the timeout instruction.

$$T(n); I \xrightarrow{\tau(m)}_{\mathcal{I}} T(n-m); I \quad \text{if } n > m \quad \text{[T-A-PASS]}$$
$$T(m); I \xrightarrow{\tau(n)}_{\mathcal{I}} \epsilon \quad \text{if } n \geq m \quad \text{[T-A-EXP]}$$
$$\underline{\alpha}; T(n); I \xrightarrow{\tau(m)}_{\mathcal{I}} \underline{\alpha}; T(n-m); I \quad \text{if } n > m \quad \text{[T-P-PASS]}$$
$$\underline{\alpha}; T(m); I \xrightarrow{\tau(n)}_{\mathcal{I}} \epsilon \quad \text{if } n \geq m \quad \text{[T-P-EXP]}$$
$$T(n); !\alpha; I \xrightarrow{\alpha'}_{\mathcal{I}} \underline{\alpha'}; [\alpha/\alpha']I \quad \text{[T-ACT]}$$
$$\underline{\alpha}; T(n); ((\pi; I) + I'); I'' \xrightarrow{\alpha:\pi'}_{\mathcal{I}} [\pi/\pi'](I; I'') \quad \text{[T-COMP]}$$

Rule [T-A-PASS] deals with time passing when a timeout is not yet expired and an action is to be made. When the timeout actually expires, rule [T-A-EXP] makes the OI moving to the error state ϵ. Rules [T-P-PASS] and [T-P-EXP] have the same purpose when a perception is expected: [T-P-PASS] deals with time passing and [T-P-EXP] makes the OI moving to error state ϵ when the timeout expires. Rule [T-ACT] removes the timeout when the agent decides to execute specialization α' of action α. Rule [T-COMP] acts the same way when a perception is made before the timeout expires.

2.2 Computational Artifacts Properties

A generic computational artifact (CA) is an artifact that is given a computational job to do on some input data, giving back either results or failure, depending on the input data. Such a computational artifact needs to be controlled during its processing by the agent it is associated with. We define the minimal usage interface as follows: possible actions are `start(input_data)`, `pause`, `restart`, `stop` and possible perceptions are `finished(result)` and `failure(error_info)`. According to the capabilities of the computational artifacts really being implemented in some application, we may extend this basic model. For example, if we want agents to have the possibility to ask the progress report of the task currently being executed or being regularly informed of the progress, the following action and perception will be added: `query_progress_report`, `progress_report`.

As we will see when trying to have several artifacts sharing the same CPU, we need mechanisms for time representation and manipulation. Timeouts proposed in [20] intuitively mean that the agent has to do an action or that a perception must occur *before* a given point in time. We also need the possibility to state that the agent has to do an action *after* a given point in time.

Hence, as for the timeout, we extend the definition of an instruction with the new atomic instruction $W(n)$. The following operational rules are added:

$$W(n); I \xrightarrow{\tau(m)}_{\mathcal{I}} W(n-m); I \quad \text{if } n > m \quad \text{[W-T-PASS]}$$
$$W(m); I \xrightarrow{\tau(n)}_{\mathcal{I}} I \quad \text{if } n > m \quad \text{[W-T-EXP]}$$

Rule [W-T-PASS] deals with time passing when the waiting time is not over. Rule [W-T-EXP] deals with the expiration of the delay imposed before being able to deal with next instruction. Note that there is no [W-ACT] rule. In the timeout case, an action can be started before the timeout has expired with rule [T-ACT]. Here, the agent cannot start an action before the end of the waiting time.

To clarify these notions, here is the example of an OI that may be used to control and interact with a computational artifact:

```
OI := W(10); !start(data);
  ((T(5); !pause; W(10); !restart; T(5); !stop)
    + ?finished(result)
    + ?failure(error_info))
```

The agent has to wait 10 time ticks before ordering the artifact to start its task with input `data`. Then, it has to order the artifact to pause within 5 time ticks and to restart no less than 10 time ticks later. Finally, if the work did not finish (normally or with an error) within 5 time ticks, the agent has to order the artifact to stop its computational task.

3. A COORDINATION ARTIFACT FOR CPU SHARING

In this section, we investigate the necessary coordination between agents to ensure meeting deadlines when they share a single CPU. We need an entity that can manage the CPU schedule and that can be used by agents to agree on the use of CPU power. As said before, this is typically where artifacts are useful. We thus define a coordination artifact called CPU Resource Sharing Artifact (CRSA) that generates operating instructions agents have to use to ensure all artifacts can meet their deadlines and which acts as an intermediary when agents negotiate CPU power.

Figure 1 reports the detailed definitions of the CRSA operating instructions. `Def_CA_Management` is the OI used by agents to interact with the CRSA when they have a computational job to give to a computational artifact, while `Def_Negotiation_Initiator` is the OI used during the previous process if the CRSA detects inconsistencies raised by adding the new computational task to the agenda. In this case, the `Def_Negotiation_Participant` OI is engaged so that other agents can propose to sacrifice a part of the CPU power they use.

Def_CA_Management. An agent which wants to delegate a computational job to a computational artifact first executes action `request(r_content)`. We consider that `r_content` is a term with four variables: an identifier `id`, an estimation `load` of the CPU power needed to finish the task, a list `t_c` of timing constraints the work has to respect and `ca_oi`, the OI that is the object of the request and which is bounded by the CRSA when no inconsistencies were found in the scheduling process. It waits for an answer from the CRSA

```
Def_CA_Management :=
  !request(r_content);T(r_timeout);
  ((?accepted(r_content);((r_content.ca_oi+0)||Def_CA_Management)+
   (?rejected(i_rejection);Def_Negotiation_Initiator(i_rejection));
     Def_CA_Management));Def_CA_Management

Def_Negotiation_Initiator(i_rejection) :=
  !start_negotiation(i_negotiation, i_rejection);  T(n_timeout);(
    ?negotiation_failed+
    ?negotiation_succeeded(i_negotation))

Def_Negotiation_Participant :=
  !get_sacrifice_request;(?finished+(?new_sacrifice_request(s_content);(
    T(s_timeout)+
    !refuse(s_content)+
    (!propose_sacrifice(s_content);(
       ?commit_sacrifice(s_content)+
       ?cancel_sacrifice(s_content))))));Def_Negotiation_Participant
```

Figure 1: CRSA Operating Instructions

Action	Precondition
request	B possible(r_content) & ¬B have_oi(r_content)
start_negotiation	B work_importance(r_content.id, s_content.imp) & B remain_work_load(r_content, s_content.load)
get_sacrifice_request	-
refuse	B ¬more_important(s_content, my_work)
propose_sacrifice	B more_important(s_content, my_work) & B can_sacrifice_work_load(s_content.load)

Perception	Effect
accepted	B have_oi(r_content.ca_oi)
rejected	-
negotiation_failed	B ¬possible(r_content)
negotiation_succeeded	B have_oi(i_negotiation.r_content)
new_sacrifice_request	-
commit_sacrifice	-
cancel_sacrifice	-

Figure 2: Semantic link between CRSA OI and agents mental state

for `r_timeout` time ticks. Answer can be an acceptance or refusal. In first case, the agent can decide or not to follow the OI `r_content.ca_oi` created by the CRSA. When a refusal is received, the agent tries to negotiate with other agents by following the OI `Def_Negotiation_Initiator`. In each case, a new call to `Def_CA_Management` is made so that the agent can do a new computational job request.

Def_Negotiation_Initiator. To simplify the negotiation process, we consider that agents have a common means of expressing the importance of a computational work, so that an agent that receives a proposal to sacrifice some CPU power can compare the importance of its works with the importance of the work it is asked to sacrifice for. The term `i_negotiation` contains the information of `r_content` and the information `imp`, calculated by the agent, that represents the importance of the requested work. The agent simply executes action `start_negotiation` and waits for an answer for `n_timeout` time ticks. If negotiation succeeds, the CRSA generates an OI to control the computational artifact and binds it to `ca_oi` in `i_negotiation`. Note that the initiator does not have any view on the negotiation process: is it managed by CRSA internal behavior.

Def_Negotiation_Participant. A participant to the negotiation protocol executes in a recursive way the action `get_sacrifice_request`. The perceptions associated to this action are `finished`, indicating that the participation of the agent to the protocol is over; and `new_sacrifice_request`, indicating that a new sacrifice request `s_content` is available. The term `s_content` contains information from `i_negotiation` and information `s_result`, calculated by the participant, on the sacrifice it can make. When a new sacrifice request is received, the agent has `s_timeout` time ticks to answer. It can refuse to sacrifice itself or it can accept, providing a proposal that will be used by the CRSA to verify if, with the sacrifice, the new task could be allocated. De-

pending on the case, the CRSA asks the agent to commit or to forget the proposed sacrifice.

Operating instructions, given alone, are not sufficient to agents. This is indeed necessary to give to agents enough information to understand the OI. We thus provide a link between the operating instructions and the agents mental state. Figure 2 describes this link in the case of our coordination artifact CRSA. We use the mentalistic semantics to actions and perceptions described in [20]. Preconditions in terms of agent beliefs are attached to OI actions and effects on agent beliefs to OI perceptions.

Preconditions of actions. An agent issues a request to the CRSA if it believes that it is possible to achieve the task (`Bpossible(r_content)`) and if it does not have an operating instruction to interact with the computational artifact yet (`¬Bhave_oi(r_content)`). Before starting a negotiation, it fills some fields in the `s_content` term (the importance and the needed load on CPU of its work). An agent refuses to sacrifice itself if it believes that its work is more important than the one in the request (`B¬more_important(s_content, my_work)`). On the contrary, it proposes to sacrifice itself if it believes that its work is less important. The proposition it does in this case depends on its beliefs of how much work load it can sacrifice (`Bcan_sacrifice_work_load(s_content.load)`).

Effects of perceptions. If an agent gets the perception that its request has been accepted or that the negotiation has succeeded, it believes that `r_content.ca_oi` is bound to a valid OI it can use to interact with a computational artifact (`Bhave_oi(r_content)`). In case the negotiation failed, it updates its mental state not to believe it is possible to do the requested task (`B¬possible(r_content)`).

4. CRSA BEHAVIOR SPECIFICATION

In this section, we describe the internal behavior of the CRSA. It mainly consists of a scheduling algorithm used to create the operating instructions it will return to agents. We overview the properties of such an algorithm, we give some information about how the scheduling is done and we give some examples of real generated operating instructions.

The scheduler that we need must have particular properties. It is constructed on top of the operating system scheduler. It can take into account (at its level) that several tasks can be carried out in parallel. It must manage differently agents, that are always active, and computational artifacts whose tasks have deadlines and therefore may be paused to ensure that other computational artifacts can meet their deadlines.

We consider that it is interesting to begin tasks as soon as possible. The behavior of a scheduler like Earliest Deadline First (EDF) [12], which will entirely execute the task whose deadline is the earliest and then turn to the next task, is not adapted to the resolution of a problem by various agents running different heuristics in parallel. In this kind of application, it is interesting to start the various heuristics as soon as possible and to stop the resolution as soon as one has found the required solution.

We model this scheduling problem with intervals and subintervals that split the scheduling horizon[4]. Each subinterval is characterized by the tasks which are allowed to run in it. In a subinterval containing n tasks, each task receives a share of $1/n$ of the CPU power in this subinterval. The scheduling problem is reduced to fixing the duration of each subinterval and is expressed as a linear program solved by the simplex algorithm.

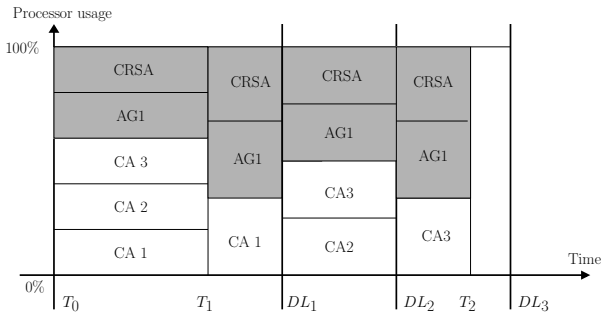

Figure 3: Schedule example

Figure 3 shows an example of such a schedule. There is an agent, the CRSA and 3 computational artifacts CA1, CA2 and CA3. Each computational artifact is assigned a task with respective deadlines DL_1, DL_2 and DL_3. We see that the main agent and the CRSA are considered to be always running (they are supposed to be fully time-aware), as they appear in each subinterval. The scheduler has determined that computational artifacts CA2 and CA3 could run until T_1 while ensuring that CA1 could respect its deadline DL_1. Here is the code of the OI generated by the CRSA for this example of schedule[1].

```
OI_CA1 := W(T0);!start(input_data);(
  (T(DL1);!stop)
  +?failure(error_info)+?finished(result))

OI_CA2 := W(T0);!start(input_data);(
  (T(T1);!pause;W(DL1);!restart;T(DL2);!stop)
  +?failure(error_info)+?finished(result))

OI_CA3 := W(T0);!start(input_data);(
  (T(T1);!pause;W(DL1);!restart;T(T2);!stop)
  +?failure(error_info)+?finished(result))
```

As this is not the main concern of the paper, we do not explain here all the details of the algorithm. The interested reader may find in [4] the algorithm details, explanations of implementation concerns and other examples of obtained schedules.

5. APPLICATION

We used the CPU resources sharing artifact previously described in a MAS which aim is to localize boats from an airplane by the sole use of the signal coming from the boat radars; it is then possible for the plane to stay hidden from the boats, which remain unaware of being spied. This procedure is called *passive localization*. It is useful in several practical situations such as the prevention of oil dumping from oil tankers – and also of course in lots of military applications.

[1]Note that, in this example, for simplicity reasons, we abuse notations for timeout and wait instructions by using absolute values of time points instead of durations as arguments to `T` and `W`. For example, we would rather use `T(`$DL_1 - t_\alpha$`)` instead of T(DL_1) in `OI_CA1`, with t_α being the absolute time the last action was taken at.

The plane (the agent) gets the bearing of the boats on a regular basis corresponding to the periodicity of the boat radars. The bearings measured are imprecise of several degrees and possibly missing, due to the harsh operating conditions which generally prevail. This approach is comparable to goniometry but boats and plane are moving, which makes things really more difficult. The computation of a location for the plane is only possible when the plane is faster than the boats, which hopefully is generally the case. This speed difference is exploited by an interval propagation algorithm on continuous domain [10] which progressively narrows the possible area of the boats in combining measurements as and when they arrive. Interval propagation is especially well adapted for this task since it allows the incremental adjunction of constraints at each new measurement and also the use of non-linear constraints, in our case trigonometrical formulas.

Time-awareness is very important for an agent using such an algorithm since the duration of the computation might change dramatically due to possible slow convergence occurring in the propagation process [11]. Actions must be taken to avoid blocking the agent for a long time. Computational artifacts are a solution since they allow the uncoupling of the cognitive part of the agent and the time-unpredictable computation tasks it must carry out. It is then possible for the agent to decide whether or not the computation must be stopped, whether or not a new measurement must be considered...

Our system uses a proprietary Prolog-based interval constraint propagator (Interlog [3]) which makes it possible to specify a time contract when initiating a resolution step. When the contract expires, if the processing is not already achieved, Interlog stops and returns the present (interval) location of the corresponding boat. The agent can decide whether to resume the propagation and possibly improve the accuracy of the localization, or to introduce new constraints depending on the arrival or not of new measurements. Assuming that the period of the radar is known (which is practically the case after few measurements) it should be possible to meet the deadline requiring that the artifact is available when a new measurement arrives. Such a requirement is relevant since a new measurement conveys a more accurate information taking into account the last position of the boat and the plane. The new measurement is thus to be considered as soon as possible.

The problem is that the plane has to monitor several boats at the same time and hence several artifacts are competing for accessing the CPU. The number of boats under consideration can be chosen by the agent in order to ensure that all the required processing is feasible. Nonetheless, because of the de-synchronization between the radars of the boats (they all have their own period and run independently), it is impossible to guarantee the meeting of the deadlines without the use of a coordination mechanism such as the one proposed by the CRSA to manage the CPU.

More formally, the problem is the following:
Let B_i be the boats, the number of which can change over time, and p_i the period of the B_i radar (it is actually an upper bound since signals might be missing but it becomes more and more certain in the course of time).

When a new measurement j arrives for boat B_i, a new task $T_{i,j}$ is created, the deadline of which is: $DL_{i,j} = T + \tau_{i,j}$, with T the time of arrival and $\tau_{i,j}$ the processing duration.

Since few is known about processing duration, the same value PD is taken and used as a time contract (in CPU time) for the interval propagator for each new task. Hence: $\forall_i \forall_j : \tau_{i,j} = PD$.

At the end of the contract, a result is always available because, thanks to the monotonicity of the narrowing procedure used by the interval propagator, there is at anytime a valid interval encompassing the actual position of the boat that can be returned.

We ran experiments in a simplified version of the application described above to exhibit the benefits of the use of the CRSA in this application. We compared results obtained in the following cases:

- *when no coordination mechanism is used to manage the CPU power allocated to computational artifacts.* There is no guarantee that the computation for a measurement will be finished before a new measurement arrives. When this is the case, the artifact continues its computation until the time contract is reached and the new measurement is ignored.

- *when the CRSA services are used.* When the CRSA accepts to schedule a task, it guarantees that the computation will finish before the new measurement arrives. If the new task cannot be inserted in the schedule, the corresponding measurement is ignored.

We calculated in these two cases the percentages of measurements that were ignored. We used the following values for the parameters presented above:

- number of boats: 10,
- p_i: 5 to 30 seconds (real time),
- PD: 2 seconds (CPU time).

In the first case, 42% of measurements are ignored. Because of the de-synchronization between the radars, there are times when many measurements arrive almost at the same time, increasing the number of ignored measurements; some time later the CPU can be idlind for some seconds, all computations being finished and no new measurement arriving. The use of the CRSA coordination mechanism brings a significative improvement, with only 7% of ignored measurements and the CPU being constantly used.

6. RELATED AND FUTURE WORKS

In this section, we explore related work in time management in intelligent agent systems with the prospect of improving the notions and (formal) models that were used in this paper.

Time management is a transverse and vast concept in MAS [7]. From our point of view, it can be split into three concerns: *(i)* temporal reasoning, in which planning and scheduling are the two topics that interest us most, *(ii)* execution and monitoring and *(iii)* the link between these two phases. Item *(i)* is pointless if nothing is done to manage point *(ii)* and the most difficult problem is to unify these two viewpoints so that the modules that manage them can interoperate.

Time management already was a key point in Agent-0 [18]: the language allows agents to schedule commitments they

have to fulfill in the future and the framework is responsible of calling the corresponding actions when necessary. In Agent-0, there is a single control loop. On the contrary, InteRRaP [13] introduced several control layers that are respectively in charge of reactive behavior, planification and cooperation. There is a clear distinction between the three points of time management we cited before. To bridge the gap between temporal reasoning and execution and to allow temporal verifications, Concurrent METATEM[6] uses agent specifications that are directly executable, but is limited to reactive agents.

Several work as also been done to manage unpredictable hard and soft deadlines that are only known at execution time. The approach proposed by Adelantado et al. [1] to this problem uses two levels: a reactive meta-level that makes the agent aware of its environment and that controls the base level, which is *anytime* based. To our point of view, their approach cannot be applied to a wide range of applications because of the difficulty to find anytime algorithms for a particular problem. Moreover, our approach is based on *continuous* meta-level reasoning, which can give more flexibility than reactive approach, and the goal of computational artifacts is to embed legacy code that can have bad temporal properties.

To deal with this problem of deadline unpredictability, Lalanda et al. [9] propose to provide run-time flexibility to agents with multiple choices to realize an action. Agents have goals with hard or soft deadlines and goals have different priorities. The agents continually reason about their deadlines to choose the next action that they will execute. This reasoning is based on a scheduler that ensures deadlines meeting and a threshold based decision process to interleave the execution of the agent plans. Lalanda et al. applied their techniques to an office robot application. The robot is given tasks as book fetching and delivery, environment learning or environment inspection. The robot can decide to degrade its performance if it cannot fulfill all its goals. Our approach is more or less similar except that the tasks of our agents are computational tasks. We view time passing as the CPU time consumed by artifacts. They compete to use the CPU and agents have to cooperate to obtain the CPU power they need to meet their deadlines.

Omicini et al. describe in [15] an extension of the ReSpecT language to manage time in coordination artifacts. An extension of the well known *dining philosopher* problem is given as example. It manages the case when philosophers do not release back the chopsticks. The final coordination policy constraints the maximum time an agent can use the shared resources. Their approach is to allow *coordination artifacts* programs to contain timed instructions (wait, timeouts, timed insertions/deletions of reactions), while our approach is to allow *agents* reason about computation time passing and cooperate to achieve their goals.

An interesting field of research to improve this work is the one initiated by IMPACT [2] and extended by Temporal Agent Programs [5]. Interesting notions like checkpoints and history could be usefully exploited in our framework. More generally, having a formal model of algorithms used in computational artifacts is of great importance. This is the idea behind anytime computation [22]. However, finding an anytime algorithm to solve real-world problem is often impossible. Finding adaptable algorithms is nevertheless the good way to follow [8, 19]. Moreover, we must go towards flexibility. Agents must not be in a situation where they are unable to do anything. They need to have several choices to respect their temporal constraints and have the means to do the right choice [21, 14].

Last, but not least, MAS are distributed in nature. We have thus to extend our CPU Resource Sharing Artifact so that it can give information to agents about the load of other CPUs in order to let them decide to migrate.

7. CONCLUSION

In this paper, we investigated one aspect of time-awareness in multi-agent systems: being able to meet deadlines on heavily computational tasks in a context of CPU sharing, while remaining aware of the environment. We showed that modeling computational tasks with agents in this context is not satisfying. We used the notion of artifact developed in [20] to introduce computational artifacts, entities that provide computational functionalities to agents that use them to achieve their goals. As this is not sufficient to ensure deadlines meeting, we also introduced a coordination artifact (the CPU Resource Sharing Artifact) which purpose is to manage the CPU agenda and to act as an intermediary when agents need to negotiate to share the available CPU power. It is worth noting that the CRSA only centralizes information on the CPU agenda but does not decide anything when a conflict appears in the schedule. Agents keep entire control on the sacrifice decision process.

8. REFERENCES

[1] M. Adelantado and S. de Givry. Reactive/anytime agents - towards intelligent agents with real-time performance. In *IJCAI'95 Workshop on Anytime Algorithms and Deliberation Scheduling*, 1995.

[2] K. A. Arisha, F. Ozcan, R. Ross, V. S. Subrahmanian, T. Eiter, and S. Kraus. IMPACT: A platform for collaborating agents. *IEEE Intelligent Systems*, 14(2):64–72, 1999.

[3] B. Botella and P. Taillibert. Interlog : Constraint logic programming on numeric intervals. *3rd Int. Workshop on Software Engineering, Artificial Intelligence and Expert Systems for High Energy and Nuclear Physics*, 1993.

[4] C. Dinont, E. Druon, P. Mathieu, and P. Taillibert. CPU sharing for autonomous time-aware agents. In *COGIS'06*, 2006.

[5] J. Dix, S. Kraus, and V. S. Subrahmanian. Temporal agent programs. *Artificial Intelligence*, 127(1):87–135, 2001.

[6] M. Fisher. Concurrent METATEM - a language for modelling reactive systems. In *Parallel Architectures and Languages Europe*, pages 185–196, 1993.

[7] M. Fisher, D. Gabbay, and L. Vila, editors. *Handbook of Temporal Reasoning in Artificial Intelligence*. Elsevier, 2005.

[8] E. Horvitz and G. Rutledge. Time-dependent utility and action under uncertainty. In *Proceedings of the 7th conference on Uncertainty in artificial intelligence*, 1991.

[9] P. Lalanda and B. Hayes-Roth. Deadline management in intelligent agents. Technical report, Knowledge Systems Lab, 1994.

[10] O. Lhomme. Consistency techniques for numeric CSPs. In *IJCAI'93*, 1993.

[11] O. Lhomme, A. Gotlieb, and M. Rueher. Dynamic optimization of interval narrowing algorithms. *Journal of Logic Programming*, 19-20, 1994.

[12] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[13] J. Muller and M. Pischel. The agent architecture InteRRaP: Concept and application, 1993.

[14] N. Muscettola. Incremental maximum flows for fast envelope computation. In *ICAPS*, pages 260–269, 2004.

[15] A. Omicini, A. Ricci, and M. Viroli. Time-aware coordination in ReSpecT. In J.-M. Jacquet and G. P. Picco, editors, *Proc. of COORDINATION 2005*, volume 3454 of *Lecture Notes in Computer Science*, pages 268–282. Springer, 2005.

[16] A. Omicini, A. Ricci, M. Viroli, C. Castelfranchi, and L. Tummolini. Coordination artifacts: Environment-based coordination for intelligent agents. In *AAMAS'04*, 2004.

[17] A. Ricci, M. Viroli, and A. Omicini. Programming mas with artifacts. In *ProMAS'05*, 2005.

[18] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[19] R. Vincent, B. Horling, V. Lesser, and T. Wagner. Implementing soft real-time agent control. Technical report, 2000.

[20] M. Viroli and A. Ricci. Instruction-based semantics of agent mediated interaction. In *AAMAS'04*, 2004.

[21] T. Wagner. *Toward Quantified, Organizationally Centered, Decision Making and Coordination*. PhD thesis, University of Massachusetts, 2000.

[22] S. Zilberstein and A. I. Mouaddib. Reactive control of dynamic progressive processing. In *Proceedings of the 16th IJCAI*, 1999.