Behaviour Management in Real-Time Agents

Carrascosa, C.; Terrasa, A.; Fabregat, J.; Botti, V.

Dept. Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n
46022 - Valencia
Spain
Phone Number: 96 3877352
{carrasco,aterrasa,jfabregat,vbotti}@dsic.upv.es

Abstract. In the same way living creatures follow different behaviours when facing different stimuli, some computing systems are able to benefit from using different computations for each significant situation faced. For example, this strategy is carried out as “working modes” in the Real-Time Systems domain and as “roles” in Multi-Agent Systems theory.

This paper presents an extension of the ARTIS agent (AA) architecture [1] allowing the agent to explicitly manage different behaviours. The extension has three main aspects: (1) the definition of the alternative behaviours for the agent, (2) the instruction of the agent on the particular conditions that should produce a behavioural change, and (3) the actual mechanisms making the change. Furthermore, since the AA is an architecture intended for hard real-time environments, all three aspects must be consistent with the strict temporal restrictions of the agent.

Overall, the behavioural extension makes the AA architecture better to deal with more complex and changing environments, effectively improving the adaptation capabilities of the agent, while maintaining its timeliness.

1 INTRODUCTION

Living creatures use to follow different behaviour guidelines when facing different stimuli. It is obvious that the more complex the creature is, the more its behaviour gains in complexity. Moreover, the ability of organisms of performing different tasks when dealing with different environments becomes an adaptive trait for them. For example, a typical predator animal wanders until it finds the prey, and then all its movements become subtle and stealthy; and, suddenly, breaks into a race for the prey. It would be a waste of resources for the predator being stealthy all the time, or maintaining top speed when wandering. Adaptation of behaviour is, hence, a key trait in the predator, as in most living organisms.

An agent can also benefit from using different behaviours. For example, in the B-Robot model [2], the Motor Coordination module is responsible for selecting an appropriate motor algorithm depending on the current situation for the agent. Particularly,
According to the importance of the task to be performed by the agent, the properties of the environment, and the required precision, this module may choose between a slow but accurate map-based navigation method or a fast but crude dead reckoning method.

Another case of changing of behaviours, in the Multi-Agent System domain, is the so-called role changes. In a typical electronic trade environment, an agent can both sell and buy, and, in each transaction, it can change between these two different behaviours, with opposed goals and probably conflicting tasks.

Finally, another domain in Computer Science on which a computer system is able to change the set of tasks it is initially configured to run is Real-Time Systems, which has coined the term mode change to denote this ability. From the perspective of this domain, a mode change is a run-time request to change the set of tasks to be run by the system by a different set, with the different sets being preconfigured off line.

The Agent/Multi-Agent paradigms and the Real-Time System domain have different reasons, and different goals, regarding this change of behaviour/role/mode. Agent and Multi-Agent systems are oriented towards achieving computing systems featuring arbitrarily complex behaviours, mimicking the actions of living creatures and communities. In such domains, the ability of dynamically modify the behaviour (or role) improves the adaptability of the system, enhancing its chances of achieving a better solution. On the other hand, in a real-time system, the ability of changing the execution mode is not to be able to find a better solution, but to devote the computing resources only for the tasks which are required at the moment. This strategy allows the system to separately guarantee the execution of the tasks which are needed in each mode (while it could not probably guarantee them all). Furthermore, the main concern in a real-time system regarding a mode change is to be able to do it while maintaining the temporal restrictions of the entire system.

The main aim of the work described in this paper is to introduce an explicit behaviour management in a real-time agent architecture called ARTIS. Thus, this work intends to join all the goals stated above, namely: to define alternative behaviours for an ARTIS Agent (AA), to instruct the agent on which are the best conditions to run in each behaviour, and to perform the behaviour changes in such a way that the strict temporal restrictions of the agent are always met. Overall, this behavioural extensions will make the AA more adaptive to changes in the problem conditions, effectively improving its abilities to find valid solutions in complex and non-deterministic environments.

The rest of the paper is structured as follows: section 2 briefly introduces Real-Time Systems, focusing on the concept of mode change. Section 3 presents the ARTIS architecture as it is now defined, after behaviours have been introduced. Section 4 explains in more detail how the meta-reasoning capabilities of the agent have been used to both instruct the agent on when to perform a behavior change and to actually do the change at run time. Section 5 illustrates the behaviour extension of ARTIS by introducing an example of an agent featuring different behaviours. Finally, Section 6 concludes and presents some future lines of work.
2 REAL-TIME SYSTEMS

2.1 Definition

A Real-Time System (RTS) is a system on which correction depends not only of the computation result, but also of the moment at which this result is produced [3]. In a RTS, some tasks have deadlines defining the biggest time interval in which the system may provide a result. If this result is achieved after this time, it won’t probably be useful. The main feature of a RTS is not to be continually interconnected or to be the quickest system. A RTS must guarantee its time restrictions and, at the same time, it must try to achieve its objectives. There are two kind of RTS [3]:

- **Soft Real-Time Systems** (SRTS): to run a task after its **deadline** just decreases the result’s quality.
- **Hard Real-Time Systems** (HRTS): to run a task after its **deadline** is completely useless, and even to not satisfy the **deadline** may have grave consequences. For this reason, a complete guarantee of the system is needed. This is normally achieved by means of an off-line feasibility (also name schedulability) analysis. This analysis checks the temporal properties of the system tasks against the computer resources of the system in the worst case scenario. This analysis is hence very pessimistic but it assures the ability of tasks to meet their deadlines under any circumstances.

A Real-Time Agent (RTA) is an agent with time restrictions. According to the previous classification, there are Hard Real-Time Agents and Soft Real-Time Agents depending on their time restrictions (if they are critical or not, respectively).

2.2 Mode Change

Due to the pessimism of the feasibility analysis in hard real-time systems, the amount of tasks that a system can guarantee can be very limited. This fact makes difficult to face very complex problems on which a lot of tasks are needed.

Some problems require all tasks to be in execution all the time, but others may have different sets at different times. In such cases, one approach is to classify the system tasks into different working modes, where only one of these modes may be active at a particular time.

If more than one mode is used, however, there is also a problem: how to make a mode change. [4] defines a mode change as a transition between two working modes in a system, each one with its own set of tasks and profile of the given tasks.

The main requirement for a mode change in a real-time system is to be schedulable, that is, to guarantee that no task loses its deadline because of the mode change. Therefore, a hard real-time system featuring mode changes has to perform an schedulability (feasibility) analysis to the tasks in each mode and to all the possible transitions among them.

A Mode Change Protocol is the mechanism the system has to do the mode change. There are two kinds of such protocols: **Synchronous** and **Asynchronous**. Synchronous protocols delay the beginning of the tasks in new mode until all the tasks in the previous mode have ended, while asynchronous protocols permit the co-existence of tasks
from both modes during the transition period. The main difference between them is that asynchronous protocols produce faster transitions but they are harder to implement and to guarantee.

3 ARTIS AGENT (\textit{AA})

This section provides a short description of the ARTIS Agent (\textit{AA}) architecture, a hard Real-Time Agent architecture (a more detailed description can be found in [1] [5] [6]), paying special attention to the way the behaviour abstraction has been incorporated and managed into this architecture (which is the main focus of this paper).

In accordance with existing agent architecture taxonomies [7], the \textit{AA} architecture could be labelled as a vertical-layered, hybrid architecture specifically designed to work in hard real-time environments [1].

The ARTIS agent architecture guarantees a response that satisfies all the critical temporal restrictions of the agent while also trying to obtain the best answer for the current environment status.

The architecture of an \textit{AA} can be viewed from two different perspectives: the user model (high-level model) [1] and the system model (low-level model) [8]. The user model offers the developer’s view of the architecture, while the system model is the execution framework used to construct the final executable version of the agent.

A toolkit called InSiDE [5] has been developed to translate the user model’ specification into the system model. This toolkit allows the developer to define the \textit{AA}’s user model and to convert this model to the corresponding system model automatically [9]. The result is an executable \textit{AA} which will run at the top of a real-time operating system.

3.1 User Model

From the user model point of view, the \textit{AA} architecture is an extension of the blackboard model [10] which has been adapted to work in hard real-time environments. This model is formed by the following elements:

1. A set of sensors and effectors allowing the agent to interact with the environment. Due to the environment’s restrictions, the perception and action processes are typically time-bounded.

2. A set of behaviours. Behaviours model the alternative ways of facing the environment which are available for the agent. At run time, the \textit{AA} is always in one behaviour, and in one only, called the active behaviour. The agent can switch to a different behaviour (i.e., make a behaviour change) when certain condition in the environment is detected.

All the behaviour changes are specified in a non-deterministic finite automaton (NDFA), whose states are the different behaviours defined for the agent and whose transitions model the available changes among behaviours. A non-deterministic automaton is needed because the agent must be able to deal with non-deterministic environments. In such scenarios, a unique behaviour sequence cannot be built off
line. On the contrary, with the agent being in a particular behaviour, many alternative situations can occur, each of which leading to a potentially different behaviour change.

In an \(\mathcal{AA}\), each behaviour is composed of a set of in-agents, each of which dealing with part of the environment (or solving part of the problem) that the agent is facing. The main reason to split the whole problem-solving method into smaller entities is to provide an abstraction which organizes the problem-solving knowledge in a modular and gradual way.

Each in-agent periodically performs an specific activity related to the solving of a particular subproblem, in such a way that all the in-agents (in a given behaviour) cooperate to solve the entire problem. Cooperation is mainly achieved by sharing of the results computed by the different in-agents, in a global memory fashion (explained below).

In-agents are classified in critical and noncritical, depending on having strict temporal restrictions or not. The execution of critical in-agents is guaranteed (by means of an off-line feasibility analysis) under any run-time circumstances. For this reason, their are devoted to solve the essential problems of the agent. In this way, the agent is assured to always achieve a minimal quality solution which maintains the problem/environment under control. Noncritical in-agents are in charge of solving the non-essential problems of the agent. Noncritical in-agents have not their execution guaranteed, but the agent tries to run as many of them as possible with the objective of maximizing the global solution quality.

In-agents exhibit some temporal and functional features which are now explained. These features are different for critical and for noncritical in-agents.

From a temporal point of view, a critical in-agent is characterized by a period and a deadline. In this way, the available time for the in-agent to obtain a valid response is strictly bounded and between the time it is (periodically) released and its deadline. In this interval, the in-agent has to provide a good-enough response to its subproblem, given the current situation of the environment.

From a functional point of view, a critical in-agent consists of two layers (see Figure 1):

- The reflex layer assures a minimal quality response in a guaranteed execution time. As explained above, an off-line feasibility analysis mathematically proves that the reflex layer of all critical in-agents (in each possible behaviour) will have enough computation resources to complete before their deadlines. The reflex layer of all the in-agents in the current behaviour make up the \(\mathcal{AA}\) ’s current mandatory layer.

- The real-time deliberative layer tries to improve the response achieved by the reflex layer. At run time, the agent uses all the free processor time to run the deliberative layer of as many in-agents as possible (this is done by using specific techniques from the Real-Time community, called “slack stealing algorithms” [11]). The real-time deliberative layers of all the in-agents in the current behaviour form the \(\mathcal{AA}\) ’s current optional layer.

On the other hand, a noncritical in-agent is temporally characterized by a period, but it does not have a deadline. Functionally, such in-agent consists of a deliberative layer only.
3. A set of **believes** comprising a world model (with all the domain knowledge which is relevant to the agent) and the internal state, that is, the mental states of the agent. This set is stored in a frame-based blackboard [12] which is accessible by all the in-agents. The developer may also specify that some significant changes in a believe should produce an event at run-time.

4. A **Control Module** that is responsible for the real-time execution of the in-agents that belong to the current behaviour of the \( A_A \). The temporal requirements of the two layers in each in-agent (reflex and deliberative) are different. Thus, the Control Module must employ different execution criteria for each one. In fact, the Control Module is divided into two sub-modules [6], Reflex Server –RS– and Deliberative Server –DS–, in charge of the reflex and deliberative parts of the \( A_A \) respectively. The two parts of the Control Module work in a coordinated way and coherently with the schedulability analysis.

The Control Module incorporates a **Meta-Reasoning** process to adapt the \( A_A \) reasoning process to changing situations. The specification of this process is the only part of the Control Module that is dependent of the application. This specification is composed by a set of meta-rules written by the \( A_A \)’s designer in the language designed to this function (as will be explained in section 4).

![Fig. 1. ARTIS Agent architecture: User’s Model](image)

It is necessary to highlight that once the specification of the previous parts is completed, it is needed to validate it to be considered an \( A_A \) user model specification. This validation must guarantee the agent’s hard real-time restrictions. It is made by means of an off-line analysis of the specification (of each different behaviour and their transitions). This analysis is based on well-known feasibility analysis techniques in the RTS community, and it is described in [9].

In this point, the definition of different behaviours for the agent has been presented. The instruction of the agent on the particular conditions that should produce a behavioural change have been introduced (though they will be detailed in section 4). The following sections will introduce the actual mechanisms making the change.
3.2 System Model

The system model provides a software architecture for the AA that supports all the high level features expressed in the user model. The main features of this model (translating the corresponding parts of the user model) are [9]:

1. A library to access to hardware devices and/or the hardware devices themselves. These correspond to the sensors and effectors expressed in the user model.
2. A set of working modes that, along with a Mode Change Protocol, implement the behaviour management expressed in the user model.

The concept of working mode has to be based on a particular task model that guarantees the critical temporal restrictions of the environment. Hence, the user model’s in-agents are translated into the system model’s tasks. Each working mode will have the tasks corresponding to the in-agents defined in the behaviour related to that mode.

According to this task model, a each task may have three parts:

- Initial: It is in charge of checking the system state with regards to the subproblem it knows. It also calculates a first answer to its problem, (probably with a low quality), but in a bounded time. So, it is the translation of the reflex layer in the corresponding in-agent.
- Optional: It improves the quality of the answer calculated by the initial part, but this improvement may use time-unbounded methods. In this way, it is the translation of the real-time deliberative layer in the corresponding in-agent.
- Final: It carries out the best answer calculated by the initial or the optional part.

According to this model, only the initial and final parts of the critical tasks (the ones translated from critical in-agents) will have hard real-time restrictions, and also will be in charge of the interaction with the environment (by means of the sensor and effectors).

The current AA implementation uses a synchronous mode change protocol (as defined in Section 2.2). This kind of mode changes makes easier the transitions between modes because tasks belonging to different modes cannot be active at the same time.

3. A shared memory (implementation of a blackboard model) with all the data corresponding to the different believes specified in the user’s model. This memory may be accessed by all the tasks.
4. The two modules in user model’s Control Module are here two separate entities [6]:

- The RS includes the real-time task scheduler (or First-Level Scheduler –FLS–). The FLS uses a real-time dispatching policy which, at run time, decides the task to be run next. This policy is compatible with off-line feasibility analysis. Having an on-line scheduler helps the AA to adapt itself to environment changes, as well as to take advantage of the tasks using less time than their worst-case execution time.

The FLS also implements the mode change protocol introduced to the AA, enabling it to change its working mode according to the transitions specified in the user model. Such changes are triggered at run time by specific mechanisms which are explained in Section 4.
– The DS includes the Second-Level Scheduler (–SLS–). This scheduler is in charge of executing the optional parts of the active tasks in the current mode.

The Meta-Reasoning management, which will be commented in the next section, has also been included as part of duties of both the RS and the DS modules.

5. Although they do not have their counterparts in the user model, the system model also includes the following elements:

– An off-line schedulability analysis of the different working modes. It is important to note that this analysis does not build a plan comprising the task execution sequence. On the contrary, the analysis only ensures the ability of the on-line dispatcher (FLS) to execute the real-time tasks guaranteeing their deadlines.

– Some slack extraction method to on-line calculate the available time for the SLS to execute the optional parts of tasks belonging to the current working mode. In slack intervals, the SLS will apply another on-line scheduling policy which will attend quality criteria to choose the next optional part to run.

4 META-REASONING MANAGEMENT

4.1 Meta-Rule Language

As it has been commented before, there is a Meta-Reasoning process integrated in the Control Module of the AA. The Meta-Reasoning capabilities of the AA are specified by means of the so-called Meta-Rule Language in the user model. This language has been defined to instruct the AA on the particular conditions that should produce some meta-reasoning actions.

One of the most important meta-reasoning actions (but not the only one) is a change of behaviour. The meta-rules which actions comprise a change of behaviour form the transitions in the behaviour non-deterministic finite automaton. In particular, the developer must specify one meta-rule for each different transition in the automaton.

Meta-rules are grouped into sets called Attention Focuses, allowing the activation (and deactivation) of such groups of rules. The skeleton of an attention focus composed by one meta-rule is presented:

```{AttentionFocus <AFName>
  {defMetaRule <Name> <Importance>
   (<EventType> <slotId>)
   (<condition>)*
   =>
   (<action>)+
  }
}
```

As can be observed in the previous meta-rule skeleton, the syntax of the meta-rule language is CLIPS-like. The meta-rule above shows that meta-rules are triggered to answer significant events (usually associated to changes in the believe set). Moreover, the rule can feature an additional condition to be satisfied before executing its associated action(s).
Although it is not indicated in the meta-rule, when the designer specifies an event, he associates an importance to such event. There is also an importance associated to each meta-rule to prioritize between meta-rules triggered by the same event and belonging also to the same attention focus.

There are two different meta-reasoning actions associated to activate and de-activate an attention focus. When activating an attention focus an attention degree is associated to the activated focus.

4.2 Event Management

As part of the system model, the meta-rules are stored in a shared memory accessed only by the Control Module.

The Control Module may dedicate part of its time to event management (see [6] for details). So, it dedicates this time to answer to pending events in decreasing importance order. To each one of these pending events, it searches in the meta-rule store for the most important meta-rule of an active attention focus with greater attention degree which event is this pending event and its associated condition is fulfilled.

This approach allows the designer to specify the meta-reasoning process in the design time that will allow the AA to adapt to significant changes in its situation, where this situation is given not only by the environment state but also by its internal state.

5 EXAMPLE: MAIL ROBOT

An ARTIS Agent has been designed to control a Pioneer2 robot, the functionality it offers is the mail delivering in a known office environment. This behaviour allows it to visit the rooms where it has a letter to deliver, taking into account the urgent ones and their deadlines and trying to choose the best path.

The AA is composed of four different behaviours usable in solving this problem:

- Waiting for new mail. The waiting for new mail behaviour is the initialization behaviour, and also the one used when all mail has been delivered. When waiting for new mail, the agent only monitorizes the charge level of the batteries and waits for new work to arrive.
- Mail dispatching. Mail dispatching is the behaviour that he AA follows when has mail to hand over. It plans the route, goes to the offices, avoids obstacles and keeps an eye on charge level and new mail. It’s the most complex behaviour in the agent, and also the most often used one.
- Mail delivering. When the agent arrives to an office, the behaviour to follow is the mail delivering. If the door is closed, it knocks by bumping twice on it. The behaviour is the same as before but we pay no attention to obstacles, because we want to collide.
- Return to base to recharge batteries. Return to base to recharge batteries, if charge level goes below a threshold, it leaves all work and returns home to sleep.

All these behaviours and the changes among them comprise the non-deterministic finite automaton of the figure 2.
As an example of the use of the meta-rules, the transition between the \textit{Mdi} and the \textit{Mde} behaviours towards the \textit{RtB} (Return to Base) means that when \textit{Mail Dispatching} or \textit{Mail Delivering}, if batteries are found to be under a trigger level, the robot should return to base to recharge. The meta-rule will be written as:

\begin{verbatim}
(AttentionFocus Emergency
 (defMetaRule BatteryLevelChecker 0
  (MODIFICATION battery.current.level)
  (battery.current.level < 10)
  =>
  (DeActivateFocus Emergency)
  (SetBehaviour RtB)))
\end{verbatim}

That means an Attention Focus is defined to deal with emergency situations, wherein we put a rule. After the name of the rule (BatteryLevelChecker) comes the priority, zero, thus, maximum. The KDM event which the rule links is the modification of the slot where the battery level is stored. The left part of the rule checks then if it’s below the trigger. If the rule fulfills its left part, the Emergency Attention Focus is deactivated, hence not triggering again in the next iteration of the control module, and the \textit{RtB} behaviour is set.

\section{Conclusions}

In this paper, an extension of the $A,A$ to manage different behaviours has been presented. This extension is carried out as a big change in all the $A,A$ structure and abstraction levels. So, it changes the user model and the way the agent must be designed. It changes also the system model incorporating mode changes management.
This behaviours management is instructed by means of a meta-rule language that allows the agent to adapt to significant situations changes, being one way of adaption to change the current $\mathcal{AA}$ ’s behaviour.

The extension presented has propelled the adaptivity feature of the $\mathcal{AA}$ architecture to another level enabling it to face problems with a complexity degree impossible to face in the previous approach.

On the other hand, although nowadays all the meta-rules are written at design time, they are translated into data structures prepared to incorporate in the future a learning algorithm able to learn and forget meta-rules.

References