Agent Reactive Capabilities in Dynamic Environments

Jorge Agüero, Miguel Rebollo, Carlos Carrascosa, Vicente Julián[1]

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València
Camino de Vera S/N 46022 Valencia (Spain)
{jaguero, mrebollo, carrasco, vinglada}@dsic.upv.es

Abstract

Currently, there are many intelligent agent models that provide a number of components or abstractions to solve different types of problems. However, few of them provide abstractions or concepts that allow considering how to deal with complex dynamic environments (an environment that changes continuously in terms of available resources, global behavioural rules, etc.) In this work, we propose two abstractions that provide the developer with a new way of modeling reactive agent capabilities in dynamic environments. The first abstraction focuses on how to process the environmental stimuli as events; the second abstraction specifies how to launch tasks in response to events, which is an approach that is based on event-condition-action rules. Moreover, we present an example that is based on a call center CBR-based application, and a performance evaluation of the proposal is also provided.

Keywords:
Agent Models, Reactive Agent Architectures, Case-Based Reasoning

1. Introduction

The design of systems to solve problems in dynamic environments is a complex task. This is most evident when addressing real problems where the dynamic nature of the environment makes solutions that are appropriate at a given time and that are inappropriate for another instant in time[6]. These systems must deal with an environment that changes continuously in terms of

[1]Corresponding author: Tel.: +34 96 387 73 50; fax: +34 96 387 73 59.

its available resources, global behavioural rules, norms, etc. The environment includes multiple actors and artifacts which may frequently enter or leave the scene unexpectedly[2].

However, most of the agent design methodologies do not include concepts or abstractions that deal with aspects of dynamic environments. In these methodologies, the environment is usually an abstraction where agents are located and interact with it. Our proposal offers tools to encapsulate the cognitive capabilities of the agent in order to face a changing environment. Thereby providing the developer with abstractions to model interactions in dynamic environments. We propose to extend the generic agent-π[3] by adding task and event models in order to increase the expressiveness of the model.

The first abstraction allows the agent to know how to process repetitive stimuli coming from the environment (how to handle events that cause changes in the environment). The taxonomy allows the agent to identify all the events and the order in which events will be processed. The second abstraction allows the agent to decide how to launch actions or tasks in response to changes in the environment (in an event-condition-action model). With this proposal, we want to provide the developer with abstractions in order to obtain specialized agent responsiveness in dynamic environments. The high-level abstractions provided are platform-independent. The corresponding rules to transform them into the concepts that are related to different execution platforms (platform-dependent) must also be defined.

In order to evaluate the proposed abstractions, we have chosen a case study that consists of the implementation of a CBR system for a help-desk environment, which provides intelligent customer support. The CBR system works with automatic incidences that are related to computer errors (network-system failures and personal computer problems) in a department that helps with the problem-solving process.

2. Agent-π: an Agent Model

In [3, 4], we proposed a set of meta-models, called πVOM (*Platform-Independent Virtual Organization Model*), which were designed using the detection of common concepts in an iterative cycle consisting of a bottom-up analysis. Common elements in existing MAS methodologies (i.e., TROPOS[5], GAIA [10], OperA[7]) were identified and incorporated to the πVOM model, which is explained in detail in [4]. The main views/models of πVOM are:

Structure, *Functionality*, *Normative*, *Agent*, and *Environment*. Figure 1 shows the *Agent* model, called *agent-π* (Platform-Independent agent). This figure provides an abstract view of the main components, concepts, and the existing relationships of the *agent-π* model.

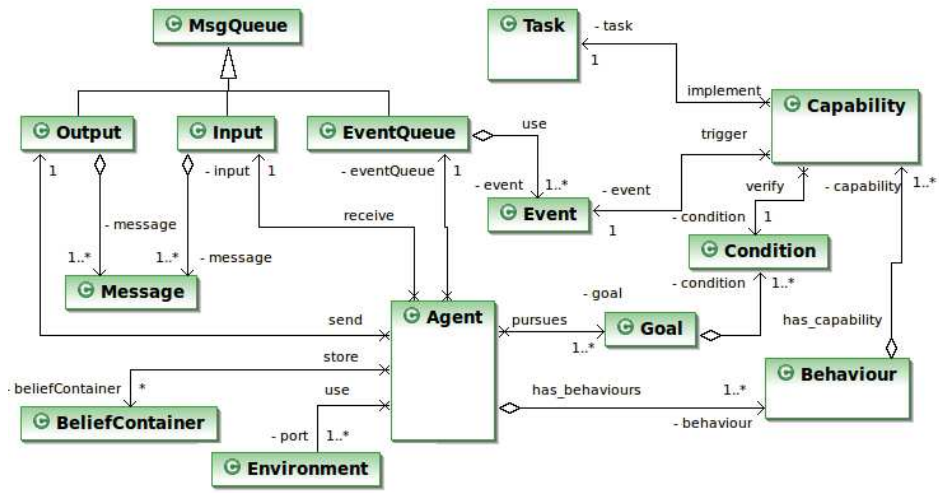


Figure 1: Concepts used in the *agent-π* model

The Agent model is a set of interrelated components, each of which serves a specific function for the definition of the agent. The main components are: *Behaviours*, *Capabilities*, and *Tasks*. The highest-level entity to be considered is the agent. The organizations, group rules, or norms (other πVOM models) are not taken into account in this paper due to space limitations.

Tasks represent the *know-how* of the *Agent* and they are the components where actions are implemented. *Capabilities* represent the different situations of the agent and control where *Tasks* are applied. *Capabilities* follow a pattern of *event-condition-action*. *Behaviours* are roles that group these capabilities. The main reason for dividing the whole problem-solving method is to provide an abstraction that organizes the problem-solving knowledge in a modular and gradual way. In the Agent model, the *Task* concept is encapsulated in a *Capability*, which is an event-oriented component that expresses the circumstances under which a *Task* must be launched for execution. A set of *Capabilities* can be encapsulated into a *Behaviour* that models the response of the agent to different situations.

3. Extending Task and Event models

As stated above, we propose two abstractions that allow highly dynamic environments to be modelled. These abstractions are implemented on the *agent-π* model (on the *Task* and *Event* meta-models), allowing the *agent-π* architecture to be extended. This new architecture proposes a novel way to model the reaction capacity of agent. This new *agent-π* provides a set of different skills (*capabilities*). This set of skills give the developer the ability to decide how to try to solve a specific problem.

3.1. Task Taxonomy

A fundamental component of the agent model is the *Task*. *Tasks* are the elements that contain the code that is associated to the agent's Capabilities. A *Task* in execution belongs to only one *Capability*, and it can be viewed as the agent's answer to a problem. However, it is the designer who must determine whether that problem must be solved only once or solved as many times as it occurs. Different instances can be activated accordingly. *Tasks* can be classified into the following types:

1. **Multiple:** Different instances of the same *Task* can be activated. For example, if a *Task* has to answer a specific message by means of an ACK (Acknowledge), the designer can decide that this answer can be done in parallel to different messages. Thus, the designer would define the *task* that is in charge of answering these messages as multiple.

2. **Exclusive:** There exists only one instance of the *Task* at a given time. This kind of *Task* can be divided into two sub-kinds according to the way new activations of instances are dealt with when the event and the suitable condition are given: (i) **Non-interrupting:** The first instance is continued until it finishes, thus delaying the possible execution of new instances of the same *Task*; (ii) **Interrupting:** The new instance eliminates the old one. For instance, if the capability is to calculate a solution to a problem and the generation of a new instance indicates that the data being used in the old instance calculation are outdated, then the calculation that is being done is no longer useful.

3.2. Event Taxonomy

An *Event* is any notification that is received by an agent informing the agent that something that may be of interest has happened in the environment or inside the agent. This may cause the activation of a new *Capability*.

In a similar way to *Tasks*, one possible classification of events comes up when the agent handles the new instances of the same event:

1. **Multiple:** There can be different instances of the same event in the queue, and all of them have to be managed.

2. **Exclusive:** There exists only one instance of an event type waiting to be attended to. Depending on the way new event instances are managed, these events can be classified into two sub-types: (i) **Non-interrupting:** If a new event instance arrives and there exists a previous instance of the same event in the queue, the new event is eliminated; (ii) **Interrupting:** If a new event instance arrives and there exists a previous instance of the same event in the queue, the old event is eliminated.

Thus, events may be generated according to different accions with different behaviors depending on the proposed taxonomies. According to the previously presented, different types of capabilities can be defined. The following section introduces the different possible combinations.

4. Capability Taxonomy

We can now analyze in detail the different types of capabilities (resulting from the combination of different types of events and tasks) that are available to the developer to solve different problems. To analyze the different features provided by our proposal, we assume that an agent A_i has various *Tasks*, *Capabilities*, and *Behaviours*, and we define the following:

1. Let $C = \{C_1, C_2, \ldots, C_I\}$ be the set of all the agent capabilities, such that the i-nth capability is $C_i \in C | i = \{1, \ldots, I\} \wedge I \in \mathbb{N}$.

2. Let E be the set of all the events that handle the agent. These events are grouped into queues $E = \{E_1, E_2, \ldots, E_K\}$, where E_K is the event queue of the capability C_k, such that $E_K \in E | K = \{1, \ldots, I\} \wedge I \in \mathbb{N}$.

3. Let $E_K[t] = \{e_k[t_0], \ldots, e_k[t_n]\}$ be an event queue at the instant of time t, containing a series of events $e_k[t_i]$ that are produced in the time interval $[t_0, t_n]$, such that $e_k[t_i] \in E_K | (0 \leq i \leq n) \wedge (t_n \leq t_i) \wedge (t_n > t_0 > 0)$

4. Let $T = \{T_1, T_2, \ldots, T_J\}$ be the set of all the agent tasks, such that $T_j \in T | j = \{1, \ldots, I\} \wedge I \in \mathbb{N}$.

5. Let $T[t]$ be the set of agent active tasks at the instant t (with $T[t] \subseteq T$).

6. Let $T_k[t]$ be an instance of the k-nth task that is launched at the instant t by the capability C_k in response to the event $e_k[t]$ of the queue $E_K[t]$ (with $T_k[t] \in T[t] \subseteq T$).

Finally, as stated above, our agent model is a process of *Event-Condition-Action* (definition 6), which is managed by *Capabilities.* However, this *Event-Condition-Action* process can be interpreted as a functional relationship, i.e., a *Capability* can be interpreted as a function $y = C_k(x)$ (a function that activates or initiates tasks). This function takes the event $e_k[t]$ as input (argument x) and produces the task $T_k[t]$ (value y) as output. Thus, an active task is launched as $T_k[t] = C_k(e_k[t])$ if the trigger condition is correct. Summarizing, this can be described as (*Capability* as a function):

$$T_k[t] = C_k(e_k[t]) = \begin{cases} \emptyset & \text{if } \textit{event condition} = false \\ T_k & \text{if } \textit{event condition} = true \end{cases}$$

With these definitions, it is possible to analyze different combinations of the proposed abstractions that allow the developer to solve different problems in dynamic environments. The implementation of this proposal assumes that the agent life cycle is determined by a set of events with a frequency f_e, which are stored in its queue. These events are processed by the agent with a rate given by the scheduler (f_s) and a task is subsequently launched in response to this event (which has a duration τ). To appreciate the usefulness of this proposal, it is assumed that $f_e \gg f_s \gg 1/\tau$, i.e., the speed with which the events arrived is very high (additional tasks may be or may not atomic). Therefore, the above description of the event and task taxonomy generates nine types of *Capabilities* that we describe in the following subsections.

4.1. mmCapability: Multiple Events and Multiple Tasks.

In this case, there is an event queue at the instant t, such that $E_K[t] = \{e_k[t_0], \ldots, e_k[t_n]\}$ (with $t_n > t_0$), and the first event in the queue is selected $e_k[t] = first(E_K[t]) = e_k[t_0]$, which is verified by the capability C_k to activate the task $T_k[t] = C_k(e_k[t_0])$ for its launch. The task and event queues (in the agent) can be described as:

$$\begin{aligned} E_K[t+1] &= E_K[t] - first(E_K[t]) \\ T[t+1] &= T[t] \cup \{T_k[t]\} \end{aligned}$$

This scenario assumes that the agent reacts to all of the stimuli in the environment and, therefore, has all the events in memory (queued). In response to this monitoring, the agent reacts by launching as many tasks as events processed.The agent has a complete monitoring of the environment and responds to all the dynamics or changes.

4.2. imCapability: Interrupting Events and Multiple Tasks.

In this case, it is assumed that there is an event queue $E_K[t] = \{e_k[t_n]\}$ that maintains only one significant event (the last one). However, we can assume that there is a real event queue at the instant t, such that $E_K[t] = \{e_k[t_0], \ldots, e_k[t_n]\}$ (with $t_n > t_0$), and in order to model interrupting events, it is only necessary to process the last event (and only one event) $e_k[t] = last(E_K[t]) = e_k[t_n]$ and then proceed to empty the queue of events that remain or are stored $E_K[t] = \emptyset$. Thus, once the relevant event is received, it is verified by capability C_k to activate the task $T_k[t] = C_k(e_k[t_n])$ for its launch. The task and event queues can be described as:

$$\begin{aligned} E_K[t+1] &= E_K[t] - last(E_K[t]) \Rightarrow E_K[t+1] = \emptyset \\ T[t+1] &= T[t] \cup \{T_k[t]\} \end{aligned}$$

In this scenario, the agent has good reactivity and launches as many new taskes as are required by relevant events. However, this scenario has a limited monitoring capability (only the last event is considered to be relevant).

4.3. nmCapability: Non-Interrupting Events and Multiple Tasks.

In this case, it is assumed that there is an event queue $E_K[t] = \{e_k[t_0]\}$ that maintains only one significant event (the first one). However, we can assume that there is a real event queue at the instant t, such that $E_K[t] = \{e_k[t_0], \ldots, e_k[t_n]\}$ (with $t_n > t_0$), and in order to model non-interrupting events, it is only necessary to process the first event (and only one event) $e_k[t] = first(E_K[t]) = e_k[t_0]$ and then proceed to empty the queue of events that remain or are stored $E_K[t] = \emptyset$. Thus, once the relevant event is received, it is verified by capacity C_k to activate the task $T_k[t] = C_k(e_k[t_0])$ for its launch. The task and event queues can be described as:

$$\begin{aligned} E_K[t+1] &= E_K[t] - first(E_K[t]) \Rightarrow E_K[t+1] = \emptyset \\ T[t+1] &= T[t] \cup \{T_k[t]\} \end{aligned}$$

This type of capability has characteristics that are similar to the previous capability. The only difference is that the event instance that is considered to be relevant is the first one received.

4.4. miCapability: Multiple Events and Interrupting Tasks.

In this case, there is an event queue at the instant t, such that $E_K[t] = \{e_k[t_0], \ldots, e_k[t_n]\}$ (with $t_n > t_0$), and the first event in the queue is selected $e_k[t] = first(E_K[t]) = e_k[t_0]$, which is verified by the capability C_k to activate the task $T_k[t] = C_k(e_k[t_0])$ for its launch. The task and event queues can be described as:

$$\begin{aligned} E_K[t+1] &= E_K[t] - first(E_K[t]) \\ T[t+1] &= \begin{cases} (T[t] - \{T_k\}) \cup \{T_k[t]\} & \text{if } T[t] \cap \{T_k[t]\} \neq \emptyset \\ T[t] \cup \{T_k[t]\} & \text{otherwise} \end{cases} \end{aligned}$$

This scenario assumes that the agent monitors all of the event instances received, but in response keeps running only one task (of a specific type). This scenario is quite reactive, but it is costly since the agent must safely stop when the task is interrupted.

4.5. iiCapability: Interrupting Events and Interrupting Tasks.

In this case, it is assumed that there is an event queue $E_K[t] = \{e_k[t_n]\}$ that maintains only one significant event (the last one). However, we can assume that there is a real event queue at the instant t, such that $E_K[t] = \{e_k[t_0], \ldots, e_k[t_n]\}$ (with $t_n > t_0$), and in order to model interrupting events, it is only necessary to process the last event (and only one event) $e_k[t] = last(E_K[t]) = e_k[t_n]$ and then proceed to empty the queue of events that remain or are stored $E_K[t] = \emptyset$. Thus, once the relevant event is received, it is verified by capacity C_k to activate the task $T_k[t] = C_k(e_k[t_n])$ for its launch. The task and event queues can be described as:

$$\begin{aligned} E_K[t+1] &= E_K[t] - last(E_K[t]) \Rightarrow E_K[t+1] = \emptyset \\ T[t+1] &= \begin{cases} (T[t] - \{T_k\}) \cup \{T_k[t]\} & \text{if } T[t] \cap \{T_k[t]\} \neq \emptyset \\ T[t] \cup \{T_k[t]\} & \text{otherwise} \end{cases} \end{aligned}$$

In this scenario, the agent only maintains the last instance of an event type that was received. Moreover, when a new event arrives, the agent stops the task for this type of event and launches a new task. This new task will be the task that best matches or adjusts the new event received.

4.6. niCapability: Non-Interrupting Events and Interrupting Tasks.

In this case, it is assumed that there is an event queue $E_K[t] = \{e_k[t_0]\}$ that maintains only one significant event (the first one). However, we can assume that there is a real event queue at the instant t, such that $E_K[t] =$

$\{e_k[t_0], \ldots, e_k[t_n]\}$ (with $t_n > t_0$), and in order to model non-interrupting events, it is only necessary to process the first event (and only one event) $e_k[t] = first(E_K[t]) = e_k[t_0]$ and then proceed to empty the queue of events that remain or are stored $E_K[t] = \emptyset$. Thus, once the relevant event is received, it is verified by capacity C_k to activate the task $T_k[t] = C_k(e_k[t_0])$ for its launch. The task and event queues can be described as:

$$\begin{aligned} E_K[t+1] &= E_K[t] - first(E_K[t]) \Rightarrow E_K[t+1] = \emptyset \\ T[t+1] &= \begin{cases} (T[t] - \{T_k\}) \cup \{T_k[t]\} & \text{if } T[t] \cap \{T_k[t]\} \neq \emptyset \\ T[t] \cup \{T_k[t]\} & \text{otherwise} \end{cases} \end{aligned}$$

This type of capability has characteristics that are similar to the previous capability. The only difference is that the event instance that is considered to be relevant is the first one received. This capability stops the previous task (which no longer responds to the event) and launches a new task.

4.7. mnCapability: Multiple Events and Non-Interrupting Tasks.

In this case, there is an event queue at the instant t, such that $E_K[t] = \{e_k[t_0], \ldots, e_k[t_n]\}$ (with $t_n > t_0$), and the first event in the queue is selected $e_k[t] = first(E_K[t]) = e_k[t_0]$, which is verified by the capability C_k to activate the task $T_k[t] = C_k(e_k[t_0])$ for its launch. The task and event queues can be described as:

$$\begin{aligned} E_K[t+1] &= E_K[t] - first(E_K[t]) \\ T[t+1] &= T[t] \cup \begin{cases} \emptyset & \text{if } T[t] \cap \{T_k[t]\} \neq \emptyset \\ \{T_k[t]\} & \text{otherwise} \end{cases} \end{aligned}$$

In this case, the agent stores all events, but these events do not launch a new task instance if there exists a previous instance of the same task still in execution. In other words, the event is not treated until the task is finished.

4.8. inCapability: Interrupting Events and Non-Interrupting Tasks.

In this case, it is assumed that there is an event queue $E_K[t] = \{e_k[t_n]\}$ that maintains only one significant event (the last one). However, we can assume that there is a real event queue at the instant t, such that $E_K[t] = \{e_k[t_0], \ldots, e_k[t_n]\}$ (with $t_n > t_0$), and in order to model interrupting events, it is only necessary to process the last event (and only one event) $e_k[t] = last(E_K[t]) = e_k[t_n]$ and then proceed to empty the queue of events that remain or are stored $E_K[t] = \emptyset$. Thus, once the relevant event is received, it is verified by capacity C_k to activate the task $T_k[t] = C_k(e_k[t_n])$ for its launch. The task and event queues can be described as:

$$\begin{aligned} E_K[t+1] &= E_K[t] - last(E_K[t]) \Rightarrow E_K[t+1] = \emptyset \\ T[t+1] &= T[t] \cup \left\{ \begin{array}{ll} \emptyset & \text{if } T[t] \cap \{T_k[t]\} \neq \emptyset \\ \{T_k[t]\} & \text{otherwise} \end{array} \right. \end{aligned}$$

Similarly to the previous capabilities, only one task instance is allowed to be in execution. Moreover, in this case, only the last instance of an event type is stored until treated.

4.9. nnCapability: Non-Interrupting Events and Non-Interrupting Tasks.

In this case, it is assumed that there is an event queue $E_K[t] = \{e_k[t_0]\}$ that maintains only one significant event (the first one). However, we can assume that there is a real event queue at the instant t, such that $E_K[t] = \{e_k[t_0], \ldots, e_k[t_n]\}$ (with $t_n > t_0$), and in order to model non-interrupting events, it is only necessary to process the first event (and only one event) $e_k[t] = first(E_K[t]) = e_k[t_0]$ and then proceed to empty the queue of events that remain or are stored $E_K[t] = \emptyset$. Thus, once the relevant event is received, it is verified by capacity C_k to activate the task $T_k[t] = C_k(e_k[t_0])$ for its launch. The task and event queues can be described as:

$$\begin{aligned} E_K[t+1] &= E_K[t] - first(E_K[t]) \Rightarrow E_K[t+1] = \emptyset \\ T[t+1] &= T[t] \cup \left\{ \begin{array}{ll} \emptyset & \text{if } T[t] \cap \{T_k[t]\} \neq \emptyset \\ \{T_k[t]\} & \text{otherwise} \end{array} \right. \end{aligned}$$

In this scenario, the agent has a behavior that is similar to the behavior in previous capability, however, the agent only stored the first instance of an event type until treated. If the agent receives similar event instances, only the first received is stored and the rest are discarded.

5. Case study

In this section, we will use an example in the domain of a customer support application to illustrate and test the proposed taxonomy. Specifically, we consider a multi-agent system that acts on behalf of a group of technicians that must solve problems in a help-desk environment that was presented in [8]. The system control every process involved in providing technological and customer-support services to an organization by means of a call center. Each agent that represents an operator of the help-desk implements a CBR module to solve each ticket. Each agent has its own knowledge resources to generate a solution for the ticket and has a Domain-CBR engine that makes queries to its domain-case case-base. The CBR engine acts as an intelligent module for the agent, helping it with the problem-solving process. Agents must solve the problems that the call center receives, which are commonly known as *tickets* in call-center jargon. In addition, the organization has signed a *Service Level Agreement (SLA)* with the customers that have contracted the support service. If the proposed solution exceeds this allotted time, the organization can receive a penalization, which becomes greater as time passes.

The system proposed in [8] has been extended to include the proposed taxonomy. For automatic management of the system, tickets that are generated automatically in response to computer errors are treated as ”events”. These events trigger the different capabilities of designed agents and launch the appropriate task instances to solve the incidences that occur raised in the different tickets. Both events and tasks (and, therefore, capabilities) are defined according to the proposed taxonomy. By default, all the tickets generated by the customers are controlled by a *mmCapability*, which captures all the events of the same type generating as many task instances as needed. Nevertheless, this behavior is not appropriate in all situations. This extension improves some problematic situations that arose in the previous version of the system, which include the following:

- On some occasions, the same task was repeatedly executed if a customer generated tickets of the same type repeatedly. This situation can easily be treated if the associated capability of the agent is defined as an *iiCapability*. In this case, the system automatically only takes into account the last generated event. Moreover, if there exists a task that is still running in an attempt to solve the same incidence, this task must be cancelled and a new instance is launched.

- Similar tasks were executed to solve the same problem when different customers launched different tickets for a network failure, with the subsequent loss of performance. This situation can be controlled by defining a *nnCapability*. This capability allows modeling situations where multiple automatic tickets/events generated by a network failure must be controlled by a unique instance of a task that must take into account the time when the failure was detected.

- In the older version of the system, a failure in the temperature of the server room generated continuous events that had to be treated by the system. In this new version, this event is controlled by an *inCapability*, which allows a non-interrumpible task instance to be launched only taking into account the last event received.

- Another problem was produced when the resolution of different incidences required the use of a limited resource (i.e., manpower). This situation caused serious problems in the previous version of the system. This situation can now be designed as a *mnCapability* which allows multiple instances of an event type to be managed and launches a non-interrumpible task instance to control the limited resource.

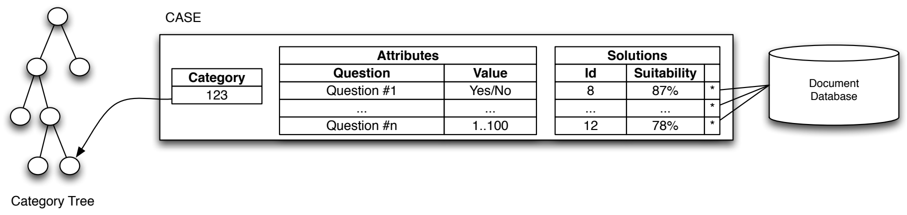
Other capability types have been introduced in the system, but due to space limitations, these are not explained here. The following subsection briefly explains how the agent employs a CBR engine to solve the incidences received in the call center.

5.1. CBR module for the problem-solving process

The CBR methodology for problem solving is typically divided into the following phases of the CBR cycle [1]: Retrieve, Reuse, Revise, and Retain. The design decisions adopted for each phase in our CBR engine were influenced by the customer-support domain and our goal of providing flexibility.

In our CBR engine, a case is the representation of a set of tickets that have the same features and the same successfully applied solutions. Each case has a set of attribute-value pairs (variables of any value type) that describe the characteristics of the problem. An indexing schema hierarchically organizes the cases in the computer memory to facilitate retrieval. The indexing process in our system consists of establishing a set of categories for each ticket that classifies it as belonging to a certain type of problem, which is currently being done manually. These categories are also stored in each case as attributes. Figure 2 shows the structure of the case taking into account the previous categorization, the set of questions and the list of possible solutions to employ.

The main goal of the *retrieval* phase is to obtain the set of stored cases that are similar to the new one. The CBR module must be able to work with heterogeneous tickets and can also be able to compute the similarity among them. Moreover, the ticket attributes may have missing values. For the retrieval algorithm, we have adapted and tested several known distance measures in order to work with heterogeneous data. The most similar case



Attributes	
Question	Value
Question #1	Yes/No
...	...
Question #n	1..100

Solutions		
Id	Suitability	
8	87%	*
...	...	*
12	78%	*

Figure 2: Structure of a case

or cases are selected by means of a k-nearest neighbor algorithm using these distance measures to compute the similarity among cases. Currently, we have implemented a Normalized Euclidean-based similarity measure and a similarity measure that is based on the ratio model proposed by Tversky [9] as shown in Eqs. (1) and (2):

$$EuclideanSimilarity(a,b) = \frac{1}{1 + \sqrt{\sum_{i=1}^{N} w_i^2 distance(a_i, b_i)^2}} \tag{1}$$

where a and b are two cases of the case-base and $w_i \in [0,1]$ is a weight assigned to each attribute i of the cases in order to indicate its importance.

$$TverskySimilarity(a,b) = \frac{\alpha(\#commonAt)}{\alpha(\#commonAt) + \beta(\#differentAt)} \tag{2}$$

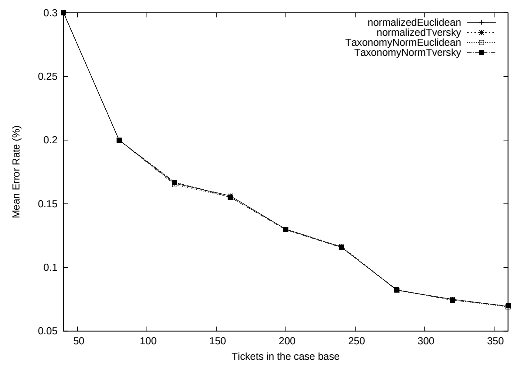
where #commonAt and #differentAt represent the number of similar and different attributes of two cases and α and β are the corresponding weights assigned to each group.

Once the most similar case is selected, its solution (the task to be executed) is proposed to solve the new problem that has been reported to the system in a simple *reuse* phase. If there exists different solutions, these can be ordered using the *suitability degree*. In this work, we have employed only the solution with the highest degree, but it is possible to propose the whole list of suitable solutions. After this, the solution is copied and applied to the new problem without any changes, but it is also configurable and any adaption process can be introduced if necessary. When a solution has been provided, the customer must indicate to the system whether that solution has really solved the problem. By means of the *revision* and *retention* phases, the system can learn the degree of suitability of the responses that the CBR engine provides. To do this: (i) the system receives an input about the used solution; (ii) if the solution applied to the ticket is already associated with the retrieved case, the suitability degree of the solution is increased; (iii) otherwise, the solution is added to the list of solutions of the case. Finally, if there is no case that is similar enough to the new ticket, the system stores this ticket and its solution in the case-base.

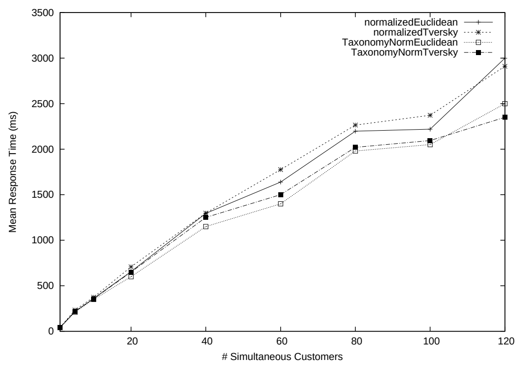
5.2. Evaluation

We have performed tests to evaluate the performance of the proposed taxonomy, comparing the extended customer support application with the earlier available version without the proposed taxonomy. In order to do the evaluation, we used a synthetic database of tickets that came from computer errors. Note that a ticket of this database is not equivalent to a case, since a case is the prototyped representation of a set of tickets with the same features and the same solution that has been applied satisfactorily in the past. We used a ticket database to test the system performance. This performance may be influenced by the number of tickets processed by the system or the number of customers sending requests simultaneously. The tests were performed using a cross-partition technique dividing the ticket database into two databases for training (loading the case-base) and testing the system. We repeated the tests for each similarity measure implemented in order to analyze its behavior in the computer-error domain.

Figure 3a shows that as the number of tickets in the case-base of the CBR system increases, the mean error in the answers provided by the system decreases. This fact demonstrates that the system knowledge goes up as the amount of processed data increases and therefore the system is learning effectively. In this case, there is no difference between the tests with or without the capability taxonomy. It can be observed that including the taxonomy maintains the effectiveness of the sytem while at the same time obtaining a better performance. This can be observed in Figure 3b, which shows how the response time is improved when the number of customers making simultaneous requests increases. As expected, the time increases in proportion to the number of simultaneous customers. In addition, this test shows that the system is able to answer the requests quickly, when the capability taxonomy is employed, even when there is a considerable number of simultaneous requests. The results of the tests also show that all the



(a) Influence of the processed data on the CBR-TM performance



(b) Influence of the amount of simultaneous customers on the CBR-TM performance

Figure 3: Evaluation Results

similarity measures behave in a very similar way and any of them might be suitable for this domain.

These results show that using the taxonomy: (i) improves the tickets/events reception management; (ii) avoids the repetition of the execution of similar tasks; (iii) uses the most appropriate information from the environment at each moment; (iv) increases flexibility in the design; and (v) obtains a better response time that facilitates compliance with SLAs, avoiding penalties and improving the quality of the service.

6. Conclusions

This work presents a *Capability* Taxonomy that provides the developer with a new way of modeling agent reactive capabilities in dynamic environments. Thus, with this *Capability* Taxonomy, the changes in the environment are not only solved by using the cognitive abilities of the agent, they can also be solved by using the two abstractions (events and tasks) that provide agents with different capabilities of reaction. The agent knows how to process repetitive stimuli coming from the environment and can decide how to launch actions to the environment in response to changes in the environment (in an event-condition-action model). This process has been illustrated with an example and validated with a prototype where the advantages of the taxonomy employment have been tested.

Acknowledgments: This work was partially supported by MINECO/FEDER TIN2012-36586-C03-01 of the Spanish government.

References

[1] Aamodt, A., & Plaza, E. (1994). Case-based reasoning: foundational issues, methodological variations and system approaches. AI Communications, 7(1), 39-59.

[2] Abraham, A., Corchado, E., Corchado, J.M.: Hybrid learning machines. Neurocomputing 72(13-15), 2729–2730 (2009)

[3] Agüero, J., Carrascosa, C., Rebollo, M., Julián, V.: Towards the Development of Agent-Based Organizations through MDD.. In: International Journal on Artificial Intelligence Tools, Vol. 22, No. 02. (2013)

[4] Agüero, J., Rebollo, M., Carrascosa, C., Julián, V.: Developing Pervasive Systems as Service-oriented Multi-Agent Systems. In: Proceedings of MobiQuitous 2010, LNICST, Volume 73, pages 78-89 (2010)

[5] Castro, J., Kolp, M., Mylopoulos, J.: A Requirements-Driven Development Methodology. In: Proceedings of CAiSE 2001 pp. 108–123 (2001)

[6] Corchado, E., Graña, M., Wozniak, M.: New trends and applications on hybrid artificial intelligence systems. Neurocomputing 75(1), 61–63 (2012)

[7] Dignum, V.: A model for organizational interaction: based on agents, founded in logic. Phd dissertation, Utrecht University (2003)

[8] S. Heras, J. A. García-Pardo, R. Ramos-Garijo, et al. Multi-domain case-based module for customer support. *Expert Systems with Applications*, 36(3):6866–6873, 2009.

[9] Tversky, A. (1997). Features of similarity. Psychological Review, 84(4), 327-352.

[10] Zambonelli, F., Jennings, N.R., Wooldridge, M.: Developing multiagent systems: The GAIA methodology. ACM Trans. Softw. Eng. Methodol. 12(3), 317–370 (2003)