

AN AGENT INTERACTION PROTOCOL FOR AMBIENT INTELLIGENCE

Yasmine Charif and Nicolas Sabouret

Laboratoire d'Informatique de Paris 6. France.

Abstract. We propose in this paper a model for agent-based devices enhanced with reasoning and interactive features. More precisely, we present an interaction protocol that enables agents to answer received messages, according not only to their functionalities, but also to their role in the interaction they are involved into. This interaction protocol aims at making it possible for agents associated to physical appliances to dialogue and compose their functionalities so as to respond to a user requirement without any prior knowledge about other agents.

Keywords. Agent-based home appliances, Ubiquitous communication, Ambient agents dialogues, Interaction protocol, Agents functionalities composition.

1. INTRODUCTION

The aim of research efforts in areas such as ambient intelligence and ubiquitous computing is to move computational power into the environment that surrounds the user. The target of such domains comprises people who want to be supported and assisted in their everyday activities.

In this vision, Ambient Intelligence (AmI) considers the creation of smart environments that integrate information, communication and sensing technologies into everyday objects. Indeed, AmI legitimately envisions such environments since it results from the convergence of three main research domains:

- Ubiquitous Computing, which aims to integrate microprocessors into everyday objects;
- Ubiquitous Communication, which enables these objects to communicate with one another and with the user;
- Intelligent User Interface, which makes it possible for the users to control and interact with these objects in a natural way.

However, current AmI work fails to benefit from the Multi-Agent System (MAS) communities research in communication and cognition for coalition formation between “intelligent” parties. Existing applications addressing devices interaction are restricted to those parties with a priori agreements to common policies and practice. Given the view that multi-agent systems are groups of agents that interact through cooperation, coordination and negotiation to satisfy their individual or common goals, AmI and agent technology can

complement each other for efficient provisioning and management of interactive and cognitive agent-based devices for the human user needs.

Thus, our research focuses on ubiquitous communication between intelligent and reasoning agents and targets the provisioning of a **generic communication protocol** that would enable objects, based on such agents, to dialogue, exchange information and constitute an ambient network. For instance we can imagine a user that may want to programme his TV and radio to decrease their sound's volume whenever the telephone rings. This situation requires simple communication between the devices (the telephone can send a command to the other appliances). But we can also imagine situations that require more complex protocols for devices dialogues and coordination. For example, a user can ask distantly its home telephone to retrieve in the teletext the match “*Real Madrid-Bayern München*” and to record it. Typically, this request requires merging TV and the DVD recorder competencies. TV can return the information related to this match (channel, hour, duration, etc) and the DVD recorder can record it provided it has received the programme details. In such an example, TV and the DVD recorder will have to dialogue and coordinate to respond to the request.

The usefulness and need of these communications for the user assistance, comfort or ‘fun’ were raised in several works such as those of Ducatel et al (4) and Eggen et al (5) who analysed the idea of assisting the human being within a smart home environment context. Starting from these requirements, we work at providing appliances with an intelligent and interactive agent representation. To this end, we develop in our research *dialogical and reasoning* agents, i.e. agents able to interact with both the human user and other agents and that are capable to look within their program to answer to asked questions. Our work focuses on designing an **interaction protocol** enabling agents’ **dialogue** games and functionalities **composition**. In such dialogues, the agents interact in a *high-level* way. A **high-level** interaction is an interaction in which the agents are not only able to answer on requests about their functionalities, state and activity, but also to *take initiatives* when they are unable to respond to the received questions, i.e. they can return a request whenever something is unknown in the asked one, look for a help from other agents, etc. The composition is then performed in order to achieve a task by collecting

successively the answers sent during the agents' dialogues.

This interaction protocol is part of the agents' features and allows them to communicate without defining a priori the possible requests and orders that could be sent from a user or other agents. Therefore, the appliances encompassing such agent representation will be provided with a generic feature enabling them to exchange high-level interactions with a user and other agents for the user concern.

Abascal et al (1) assumes that home is the perfect place to apply Ambient Intelligent precepts and technologies for giving high-level services to the user. Therefore, we choose home as the application domain to test the dialogical and reasoning features of our agents.

The remainder of this paper is structured as follows. Section 2 analyzes the related work addressing communication issues in home appliances. Section 3 presents the reasoning and interactive properties offered by the description model on which is based our agents design. Section 4 describes the interaction protocol enabling our agents' dialogue games and composition. Section 5 demonstrates the behaviour of our algorithms in a home appliances scenario. And lastly in section 6, we discuss our work shortcomings and outline its perspectives.

2. RELATED WORK

Few work in AmI proposed models to enhance home appliances with interactive features enabling them to interact with one another in a generic way. The reason is that such a feature requires to be integrated to objects at an abstract level rather than the development one.

We can quote among research efforts which approach the issue of appliances interaction the work of Gárate et al (9). The authors developed a power-line network where home appliances are connected and managed by a central controller. The user can dialogue with his home appliances and ask the services and functionalities related to them by talking naturally. The controller has a human representation that the user can see and can interact with. When the user talks the controller extracts the different commands from those vocal orders and controls the home devices. This approach addresses the human-machine interaction. However, the authors don't expound how the interaction capabilities are implemented. They talk about an extended grammar but don't explain the transition between the user phrase formalisation and the asked component program. Moreover, the machine-to-machine interaction feature is not addressed.

Another work addressing appliances interaction is the one of Lashina (10) who gives in (10) an overview of the Intelligent Bathroom concept prototyped and

demonstrated in the HomeLab of Philips Research. It comprises an interactive mirror which display is the central interaction with the bathroom. It can interact with the devices regularly used in the bathroom. For instance, a scale, displaying the weight in the mirror, can also activate the health coach informing the user about the cardiovascular state of health and give advice on improvement. The interaction between devices is clearly addressed in this project. Nevertheless, the approach developed is centralized around the mirror, and the possible couples of interacting components are already defined. Therefore, generic components interaction is not addressed.

In Vallée et al. (14), the authors propose to compose services offered by ambient objects, using semantic web techniques, so as to respond to specific tasks required by a user. In their approach, an abstract plan is defined for each task. Their composition mechanism can then match each subtask with an existing service, taking into account context parameters and testing the coherence between the concrete services found. The main weakness of this approach is that it supposes the existence of a library of abstract plans for particular requests a user can ask.

Having in mind the above, we present hereafter a proposal that tries to address the highlighted shortcomings, namely generic interaction capabilities for home appliances implying any prior knowledge about their environment. We present a model to design and enhance the agent-based devices providing them with dialogical and reasoning features. We then describe the interaction protocol we are developing to allow agents dialoguing and composing their functionalities in a generic manner.

3. A DESCRIPTION MODEL FOR ENHANCED AMBIENT AGENTS

The design phase of our work resulted in the concept of ambient agents. In order to describe such agents, capable of interacting and answering requests according to their program behaviour, we propose to make use of a specific model that integrates these features at the conception level. In particular, this model should allow to programme autonomous agents able to provide services or specific functionalities, as home appliances do. The agents should also be able to reason about the services they offer, and to exchange high-level interactions so as to satisfy the user's request about a specific service requirement.

The formalism we employ to model our agents is VDL [Sabouret] (12), which stands for *View Design Language*. VDL is a programming language for dialogical and reasoning agents basically developed for human-computer interactions (HCI) and making it possible for human users to ask agents a wide range of questions about their functioning. We choose this model

for the interactive and reasoning features it offers to the modelled agents, and extend it with an interaction protocol so as it enables inter-agent dialogues. We summarize here its main principles.

3.1 The VDL Architecture

A VDL agent is first defined as the combination of two layers:

1. *a concrete agent* reasoning about the offered service, processing the messages receipt, building the messages answer, and managing the inter-agents communication issues.
2. *an abstract service* representing the VDL-XML declaration of the offered service, associated to an ontology defining the concepts and action names handled by the service.

Namely, VDL allows implementing agents that are able to offer semantic services. Each agent can reason about the service it provides, invoke it, answer to requests about its functionalities, state and activity, and consult its ontology for semantic inference and processing. As illustrated on figure 1, using a generated user interface, a human user can interact with the VDL agent or invoke the service provided. The human-agent interaction is ensured by the VDL request model used for the requests formalisation, whereas the inter-agent interaction is made possible by an agent communication language (ACL) where requests are encompassed in message structures.

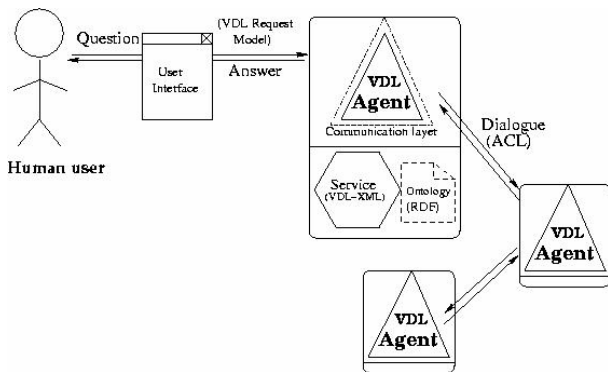


Figure 1 – The VDL Architecture.

In this paper, we focus on the interaction protocol built upon the ACL. The latter will allow us to design the interaction protocol enabling the generic agents' interaction features and thus the agents' dialogue games.

3.2 The VDL Model

The agents involved in our architecture are modelled using VDL. The VDL language (12) is implemented in

Java using concurrent threads for the individual agents, and an interpreter which provides the autonomous behaviour and the reasoning tool about the services code. The service's code is an XML tree where some nodes define actions to perform, e.g. "shut-down-oven", preconditions, e.g. "if the timer is over" or data to manipulate, e.g. "temperature". At each execution step, the agent looks within this tree for specific elements corresponding to actions descriptions. It interprets these elements so as to perform specific actions that will rewrite the tree.

Consequently, the devices based on VDL agents will be provided with a representation of their functionalities that a module can access and interpret. This **reasoning** feature, i.e. the ability for agents to read their code, is essential to allow the agents answering according to their functionalities, state and activity. The formalisation of queries and answers is made possible with the request model presented in the following.

3.2.1 The Request Model. Since the VDL request model is not the main topic of this paper, we will just give an overview of its principles.

The VDL request model [Sabouret] (11) is used to ensure requests formalisation. Initially, this model was used to represent natural questions of the user. It is now also used to express the content of the messages exchanged between several agent-based devices.

A categorisation work done on the possible questions asked by a human user to an agent (11) led to distinguish some main request ranges each request being classified according to its performative and its procedural type. A request range represents the category of the message content exchanged between a human and an agent or between two agents. The main request ranges identified are the following:

- R^{Order} is the range of commands. It represents an agent (or a user) query expressing an order to an agent (or to the service it offers) to perform an action. e.g. "Display the calling number". It is similar to FIPA requests which performative is *request* [FIPA] (6).
- R^{How} used by the agent when it has to ask for an action definition. e.g. "How do you increase-brightness?".
- R^{What} composed of:
 - R_{is}^{What} used by the agent when it has to ask for an expression value. e.g. "What is your temperature's value?".
 - R_{can}^{What} used by the agent when it has to query for another agent capabilities. e.g. "What can you do?".
- R^{Assert} is the range of assertions. It is mainly composed of:
 - R_{is}^{Assert} used by the agent when it has to send a concept or an action definition. e.g. "increase-brightness consists in testing if the screen is switched

on, then in adding 1 to the current brightness value ...”, or “record means save”.

- R_{can}^{Assert} used by the agent when it has to send an assertion on its service’s capabilities. e.g. “I can display, increase-brightness, shut-down”.
- R^{Ack} is the range of acknowledgments, used by the agent when it has to confirm that an action has been performed by the service it offers. e.g. “I’ve finished recording the program”.
- $R^{Unknown}$ used by the agent when it has to express that it does not recognize an action or a concept. e.g. “I don’t know what temperature means”.
- R^{Ask} used by the agent when it has to express a query on an XML database. e.g. “give me the channels broadcasting football matches on Friday”.

Those requests are processed by the request processing module (RPM) according to the agent functionalities, state and activity. Every VDL agent is provided with an RPM which provides part of the reasoning features of the VDL agents. Its role consists in building an answer to the received requests regarding the agent’s data structure and actions description in VDL. The answer built is also formalised as a VDL request.

Table 1 shows the processing of the main request ranges carried out at the RPM level.

Received Request	Built Answer
R^{Order}	R^{Ack} or $R^{Unknown}$
R^{How}	R_{is}^{Assert} or $R^{Unknown}$
R_{is}^{What}	R_{is}^{Assert}
R_{can}^{What}	R_{can}^{Assert}
R^{Ask}	R_{is}^{Assert}

TABLE 1 – Received requests and corresponding answers built by the RPM.

This table shows that for each received request, a processing is done or a specific answer is built by the RPM. For instance, an agent receiving an order R^{Order} executes it whenever it understands it and acknowledges the requester through a R^{Ack} request built by the RPM.

Otherwise, it sends a $R^{Unknown}$ request stating the unknown elements. This is done by the RPM which, in a general way, builds a specific answer according to the received request range and the agent’s code. To take another example, when an agent receives a R_{can}^{What} request about its capabilities, the RPM builds a R_{can}^{Assert} answer.

We presented here the RPM level in which the agent takes into account its functionalities to answer. This level represents our starting point to build the interaction protocol allowing agent-based devices to dialogue so as to achieve a task. In the interaction protocol level, the agent does not only consider its functionalities to answer to received requests but also its interactions context, i.e. the interaction goal, the surrounding agents consulted and the previous messages exchanged. The next section describes the messages structure in which VDL requests are encompassed together with the messages properties.

3.2.2 The Agent Communication Language. As we envision in our approach to enable high-level communications between the agent-based appliances, we need a communication layer in which requests are encompassed into message structures containing an *id*, the identifiers of the sender and recipient agents, etc. Hence, we need an Agent Communication Language (ACL).

The ACL we use is called *VDLMessage*. It is based on FIPA-ACL [FIPA-ACL] (7) messages (*ACLMessage*). The message content of a *VDLMessage* can be a set of VDL requests, which enables agents to express high-level queries.

The structure of a *VDLMessage* is defined using the same parameters as an *ACLMessage*, namely:

- *sender* and *receiver* Agents Identifiers (AID);
- *content* which is the message content. A *VDLMessage* content is a set of VDL requests denoted by $content(\mathcal{M}) = \langle r_1, \dots, r_n \rangle$. Part of the requests composing a *VDLMessage* can be dependant one of another as in “look for the products missing in the fridge and sort them by type”. This query for instance can be decomposed into two requests $r_1 = \text{“look for the set } P \text{ of products missing in the fridge”}$ and $r_2 = \text{“sort the set } P \text{ by type”}$ where r_2 depends of r_1 .
- the *reply-with* slot which is the unique *id* for the message;
- the *in-reply-to* slot which is the *id* of the message the involved one answers to;
- the *reply-to* slot which is the AID of the agent to which the answer must be returned.

Figure 2 shows an example in the utilisation of the *reply-with* and the *in-reply-to* slots.

In the rest of the paper, we will need a readable representation of the agents’ interactions to outline clearly the interaction protocol specifications. We choose the AUML agent interaction protocol diagrams (6) for modelling interactions between agents [Cabac] (3) (see figure 2). These diagrams are an extension of the Unified Modelling Language (UML) sequence diagrams [Booch et al] (2) but they are more powerful in their expressiveness. Moreover, they can describe a set of scenarios.

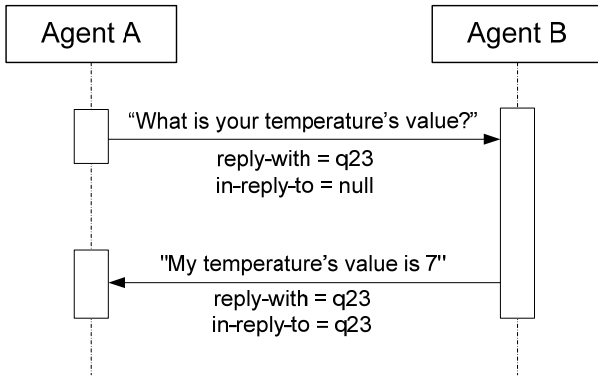


Figure 2 – An example of a VDL agents' interaction.

3.3 Summary

We presented in the above the underlying formalisms ensuring the reasoning and interactive properties of our agents. We now present an interaction model for ambient agents inspired by the multi-agent systems protocols to make it possible for agents to *dialogue* and *compose* their functionalities in order to achieve a task. We propose to enable agents keeping track of the exchanged messages, of the consulted agents and of the requests triggering the interactions. Thus, an agent can process a message regarding these information and react according to its role within the interaction, not only according to its service's code. It can then respond whenever it didn't understand an order, ask for some assistance or delegate a received request to another agent, broadcast a request to surrounding agents, etc.

4. AN INTERACTION PROTOCOL FOR SMART AGENT-BASED DEVICES

To design a generic interaction protocol for dialogical agents, we have to consider that an agent should process a received request not only according to its services functionalities, which is a feature already provided by the RPM (request processing module), but also by taking into account the previous messages exchanged, the surrounding agents and the request that triggered a dialogue between the agents. We present hereafter the technical specifications to design our interaction protocol.

4.1 A Trace for Interactions

Each agent is provided with a **history table** so as to record the information related to its interactions.

A history table \hat{h}_A , associated to an agent A , is a set of **records** ∇ , each of which is dedicated to an interaction. A set of messages are related to the same interaction if they are defined with the same *reply-with* slot's value. A record related to an interaction is defined as $\nabla \stackrel{\text{def}}{=} (id, m_0, m'_0, M, A)$, where id is the record (or the interaction) identifier, which corresponds also to the *reply-with* slot's value of the messages involved in the related interaction. m_0 is the interaction's *triggering message*. We call a **triggering message** the message that started the interaction. Such a message is characterised by *in-reply-to* = *null*. The **interaction goal** consists in satisfying the triggering requests $\{r_{01}, \dots, r_{0n}\}$ composing m_0 . m'_0 is a message containing respectively the answers $\{\rho_{01}, \dots, \rho_{0n}\}$ to each request composing m_0 . M is the set of messages exchanged, i.e. sent and received, related to the same interaction. A is the set of agent AIDs requested for resolving the triggering request.

For example, suppose that the agents A, B, C are involved in the following interaction, where each message pattern represents {sender, receiver, content, reply-with, in-reply-to}:

- $m_0 = \{A, B, \text{"What is the product RAM capacity"}, q23, \text{null}\}$
- $m_1 = \{B, C, \text{"What is RAM?"}, q23, q23\}$
- $m_2 = \{C, B, \text{"RAM is central memory"}, q23, q23\}$
- $m_3 = \{B, A, \text{"The product RAM capacity is 512Mo"}, q23, q23\}$

Hence, the agent A will have in its history table the record $\nabla = (q23, m_0, m'_0, \{m_0, m_1, m_2, m_3\}, \{B, C\})$ where the content of m'_0 is "The product RAM capacity is 512Mo".

Each time the agent receives a message, it has to store or refresh in its history table the appropriate elements of the corresponding record. And when an agent has to build an answer for a received message, it glances at its history table to respond according to the past messages sequence, to the agents already requested and to the interaction goal. The detailed specification of our interaction protocol is presented in the following.

4.2 The Interaction Protocol

The purpose of our interaction protocol is to allow agents to dialogue with each other and compose their functionalities in order to achieve a task by processing the received messages and taking initiatives when they are not initially able to answer them.

In our interaction protocol, when an agent receives a message, it first checks whether it is a triggering one or

not. Algorithm 1 and 2 describes the processing carried out by a VDL agent respectively while it receives a triggering message and a message involved in an already started interaction.

A VDL agent receiving a message has to deal with a set of requests $\{r_1, \dots, r_n\}$ composing this message. To answer such a message, the agent has to process each request r_i separately and store its answer so that it builds a set of corresponding responses $\{\rho_1, \dots, \rho_n\}$. At the end of the requests processing, the answering message is built and sent.

The algorithms below describe the appropriate answer to associate to each request range depending on the interaction context (the kind of the received message, the agent to which the answer must be returned, etc). We only consider in our interaction protocol the communication between agents. We don't take into account here the human user. Moreover, we suppose that there exists in the agent environment at least one agent that is capable of solving part of the query sent by the interaction initiator agent.

4.2.1 Processing Triggering Messages. In our interaction protocol, if an agent receives a triggering message containing an order, it has to look for the unknown elements (concepts/variables or action names) composing this order. If it recognises all the order elements, it stores an acknowledgment R^{Ack} in the built message answer. Otherwise, it stores a $R^{Unknown}$ request.

For all the other request ranges, the agent processes the request at the RPM level following Table 1.

When the agent finishes processing the requests contained in the received message, it builds for its history table the message \mathcal{m}'_0 containing only answers to the received requests which don't belong to $R^{Unknown}$. Thus, \mathcal{m}'_0 will only contain final answers to the triggering message requests.

To finish, the agent creates a new record to store the triggering message together with the information related to the interaction it will cause, if any. Moreover, it sends the built answer to the received message sender.

Let's see in detail the algorithm processing triggering messages. Let R be a VDL agent, \mathcal{m} the message it receives, \mathcal{m}' the message built in answer to \mathcal{m} ; $processRPM$ the function returning the VDL request built by the RPM according to Table 1, $unknownActions$ and $unknownConcepts$ the functions used to raise respectively the unknown action names and concepts (according the agent's code, ontology and assertions contained in \mathcal{m}), all a VDL keyword to point out the surrounding agents. The algorithm for the triggering messages processing is as follows:

Algorithm 1. *Processing Triggering Messages.*

1. **For each** request r_i of \mathcal{m} **do**

Let \mathcal{m}' be the message built in answer to \mathcal{m}

1.1 **if** $r_i \in R^{Order}$ **then**

if $knownActions(r_i)$ and $knownConcepts(r_i)$ **then**
execute the order r_i

build $\rho_i \in R^{Ack}$

else

build $\rho_i \in R^{Unknown}$

$content(\mathcal{m}')[i] \leftarrow \rho_i$

1.2 **else**

$\rho_i \leftarrow processRPM(r_i)$

$content(\mathcal{m}')[i] \leftarrow \rho_i$

2. **build** \mathcal{m}'_0 **so that:**

For each request $\rho_k \in R^{Unknown}$ of \mathcal{m}' **do**

$content(\mathcal{m}'_0)[k] \leftarrow \rho_k$

3.

3.1 **build** \mathcal{m}' **so that:**

- $receiver(\mathcal{m}') \leftarrow sender(\mathcal{m})$

- $reply-with(\mathcal{m}') \leftarrow reply-with(\mathcal{m})$

- $in-reply-to(\mathcal{m}') \leftarrow reply-with(\mathcal{m})$

3.2 **store in** \mathcal{h}_R **the new record** $\nabla = (reply-with(\mathcal{m}), \mathcal{m},$

$\mathcal{m}'_0, \mathcal{M} \leftarrow \{\mathcal{m}, \mathcal{m}'\}, \mathcal{A} \leftarrow \{sender(\mathcal{m})\})$.

3.3 **send the answer** \mathcal{m}' **to** $sender(\mathcal{m})$.

4.2.2 Processing Non Triggering Messages. Now if an agent receives a message involved in an already started interaction, i.e. a non triggering message, it has to process it considering certain context parameters.

For instance, if an agent receives a message containing $R^{Unknown}$ requests about action names from an agent S , this means that it has to include S to its set of already consulted agents \mathcal{A} . Indeed, the recipient agent knows that it is useless to ask again S since the latter already acknowledged that it is unable to recognize the main actions to perform.

Another example is an agent receiving a message containing assertions. Indeed, each time an agent receives a message containing R_{is}^{Assert} or R^{Ack} assertions, it has to store them at their corresponding index in the \mathcal{m}'_0 parameter of the agent's history table. Hence, \mathcal{m}'_0 will always contain the sum of the answers collected during the agent's dialogues.

The next step of messages processing is the one illustrating the composition feature of our interaction protocol. Indeed, after having processed all the requests composing the received message, the agent has to

update the \mathcal{m}'_0 parameter related to the interaction. We explained above that it has to include in it the assertions received from other agents. In this step, the agent has also to include the new assertions it built and stored for the message answer.

In practice, if this agent initiated the interaction and has been unable to solve the triggering message, it has to broadcast it. However, instead of spreading the initial triggering message, it replaces the solved requests by their corresponding answers stored in \mathcal{m}'_0 .

As a consequence, each agent can solve a message part using its competencies. This is done even if there exists dependencies between the requests composing it since the message is broadcasted each time it is updated with new assertions, until it is entirely solved.

To finish, the agent has to test if the triggering message has been solved and whether it is the triggering message sender or not. If the triggering message is not solved yet, it has to broadcast it by including the solved parts and by considering the agents to exclude from the recipient agents. If the recipient agent is the sender of the triggering one, the messages sequence ends.

Let's see in detail the algorithm processing non triggering messages. We note *getMessages* and *getAgents* the functions used to return respectively the parameters \mathbb{M} and \mathcal{A} of an existing record, and *paramType* the function returning the type of a request object, namely an action or a concept (or variable). The algorithm for the non triggering messages processing is as follows:

Algorithm 2. Processing Non Triggering Messages.

```

1. For each request  $r_i$  of  $\mathcal{m}$  do
  Let  $\mathcal{m}'$  be the message built in answer to  $\mathcal{m}$ 
  1.1 if  $r_i \in R^{Order}$  then
    if knownActions( $r_i$ ) and knownConcepts( $r_i$ ) then
      execute the order  $r_i$ 
      build  $\rho_i \in R^{Ack}$ 
    else
      build  $\rho_i \in R^{Unknown}$ 
       $content(\mathcal{m}') [i] \leftarrow \rho_i$ 
  1.2 else if  $r_i \in R^{Unknown}$  then
    if paramType( $r_i$ ) = TYPE_ACTION then
       $\mathcal{A} \leftarrow getAgents(\hat{h}_R, reply-with(\mathcal{m}))$ 
       $\mathcal{A} \leftarrow \mathcal{A} \cup sender(\mathcal{m})$ 
  1.3 else if  $r_i \in R^{Assert}$  or  $r_i \in R^{Ack}$  then
     $content(\mathcal{m}') [i] \leftarrow r_i$ 
  1.4 else
     $\rho_i \leftarrow processRPM(r_i)$ 
     $content(\mathcal{m}') [i] \leftarrow \rho_i$ 
2. update  $\mathcal{m}'_0$  so that:

```

For each request $\rho_k \in R^{Unknown}$ of \mathcal{m}' **do**

```

   $content(\mathcal{m}'_0) [k] \leftarrow \rho_k$ 
3.
3.1 if  $R = sender(\mathcal{m}_0)$  then
  3.1.1 if  $\rho_k \in R^{Assert}$  or  $\rho_k \in R^{Ack}$ ,  $\forall k \in \{1, \dots, |content(\mathcal{m}')|\}$  then  $\mathcal{m}_0$  is solved
  3.1.2 else
    build  $\mathcal{m}'$  so that:
    -  $receiver(\mathcal{m}') \leftarrow all / \mathcal{A}$ 
    - For each request  $\rho_j \in R^{Unknown}$  of  $\mathcal{m}'$  do
      if  $content(\mathcal{m}'_0) [j] = null$ 
         $content(\mathcal{m}') [j] \leftarrow content(\mathcal{m}_0) [j]$ 
      else
         $content(\mathcal{m}') [j] \leftarrow content(\mathcal{m}'_0) [j]$ 
    -  $reply-with(\mathcal{m}') \leftarrow reply-with(\mathcal{m}_0)$ 
    -  $in-reply-to(\mathcal{m}') \leftarrow reply-with(\mathcal{m}_0)$ 
  3.2 else
    -  $receiver(\mathcal{m}') \leftarrow sender(\mathcal{m})$ 
4.  $\mathbb{M} \leftarrow getMessages(\hat{h}_R, reply-with(\mathcal{m}))$ 
    $\mathbb{M} \leftarrow \mathbb{M} \cup \{\mathcal{m}, \mathcal{m}'\}$  and update the corresponding  $\nabla$ 
   send  $\mathcal{m}'$  if any.

```

We outlined in these specifications the several kinds of requests an agent can receive. For each case, we explained how an agent has to behave to process the received message and eventually the answer it has to send or broadcast. To have a clearer idea of this interaction protocol behaviour, we show hereafter a scenario with VDL-agent-based home appliances using our interaction protocol to communicate and assist a human user for an entertainment task.

5. A SAMPLE EXAMPLE

We present here in detail the example we talked about in section 1. It is a scenario involving a human user, a telephone, a television and a DVD recorder, each of which is based on a VDL agent provided with the interaction protocol defined above. This example will allow us to observe the behaviour of our protocol and highlight its shortcomings.

In our scenario, the human user, let's call him *bob*, is at work. *Bob* remembered that *Real Madrid* and *Bayern München*, its favourite football teams, play a match the same day. However, *bob* doesn't know on which channel and at what time this match will be broadcasted. Since this program will possibly be broadcasted before he leaves his office, he would like to record it! To do so, *Bob* sends a message to his home telephone, using for instance his mobile phone, asking it to find in the teletext for a specific program which theme is a

football match, involving *Real Madrid* and *Bayern München* teams, and to record it. Since *bob* has got a telephone, a TV and a DVD recorder at home, based on VDL agents, and if he didn't confuse about his program, his football match will be found and recorded. Figure 4 depicts how this happens.

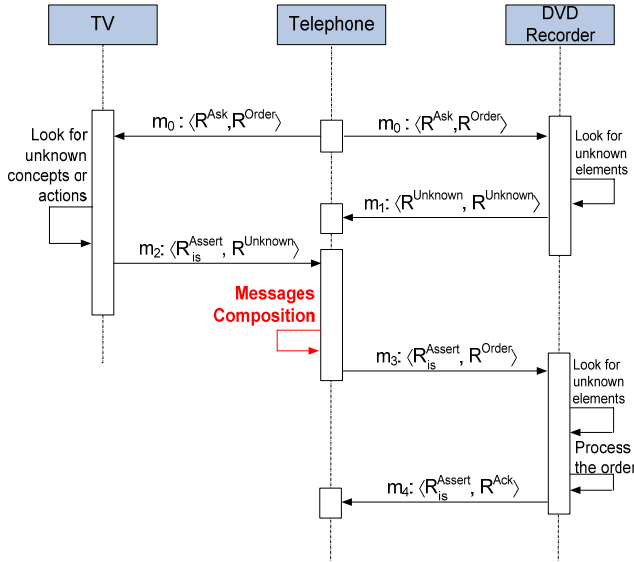


Figure 4 – Agent-based Home Appliances Example.

This figure represents the messages sequence starting from the telephone:

- The home telephone, since it is unable to answer *bob*'s query, forwards it to the surrounding agents. It encompasses the requests composing it, namely “(R^{Ask}) Find in the teletext the program *P* where play a football match *Real Madrid* and *Bayern München*, and (R^{Order}) record it” into the message m_0 .

TV and the DVD recorder receive m_0 .

- When the DVD recorder receives m_0 , it can't answer to any of its requests, since it doesn't know the teletext concept and can't perform the football match recording without having the program details. Since m_0 is a triggering message, the DVD recorder sticks to line 1.2 and 1.1 of Algorithm 1. It then sends to the telephone “($R^{Unknown}$) I don't know what is teletext and ($R^{Unknown}$) I don't know what is the program to record” encompassed in the message m_1 .

- TV receives m_0 and is capable of answering the first request. It also tries to seek in its agent's code and ontology the concepts and action names definition and fails to find the “record” definition. Following line 1.2 and 1.1 of Algorithm 1, it sends the answer “(R_{is}^{Assert}) The program *P* is broadcasted on channel6, at 4.30 p.m but ($R^{Unknown}$) I don't know what record means” encompassed in the message m_2 .

- The telephone receives m_1 containing two $R^{Unknown}$ requests. Therefore, it received no new information for solving the triggering message.

- Then the telephone receives m_2 containing a request from the R_{is}^{Assert} range. Following line 3.1.2 of Algorithm 2, the telephone builds the message to broadcast by composing the triggering message and the collected assertions (in this example, the R_{is}^{Assert} assertion about the program *P*). In figure 4, this is represented in bold by “Messages Composition”.

Moreover, since TV sent also a $R^{Unknown}$ request about an action, the telephone excludes it from the recipients (line 1.2 of Algorithm 2). Therefore, the telephone sends to the DVD recorder the query “(R_{is}^{Assert}) The program *P* is broadcasted on channel6, at 4.30 p.m, (R^{Order}) record it” encompassed in the message m_3 .

- The DVD recorder receives m_3 containing an order. It looks in the elements composing this order for unknown concepts or action names. It raises the unknown variable program *P* and finds its definition in the R_{is}^{Assert} request contained in the received message. The DVD recorder can then process the order and acknowledges the telephone using the message m_4 following line 1.1 and 3.2 of Algorithm 2.

- The telephone receives m_4 containing R_{is}^{Assert} and R^{Ack} which are respectively the awaited request ranges for the requests R^{Ask} and R^{Order} composing the triggering message. Since the initial task has been achieved, the messages sequence stops (line 3.1.1 of Algorithm 2).

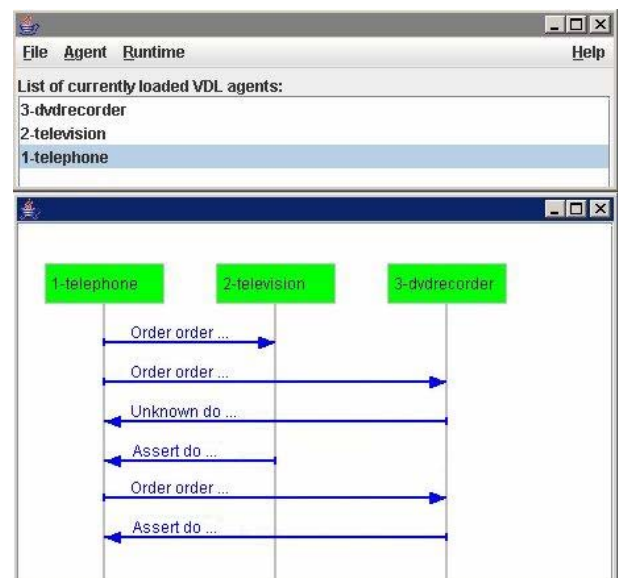


Figure 5 – A Screenshot of the Agents Interactions.

We have implemented this scenario by modelling three VDL agents: *telephone*, *television* and *dvdrecorder*. In our application, the agents executing in the same platform are stored in a list. This is what allows the home telephone (or any other agent) to send a message to the surrounding agents.

We took screenshots of the resulting agents interactions from our application. Figure 5 shows our application sniffer which is a tool representing with simple sequence diagrams the agents interactions, and figure 6 represents the message m_2 sent from TV to the telephone and containing the requests R_{is}^{Assert} and $R^{Unknown}$.

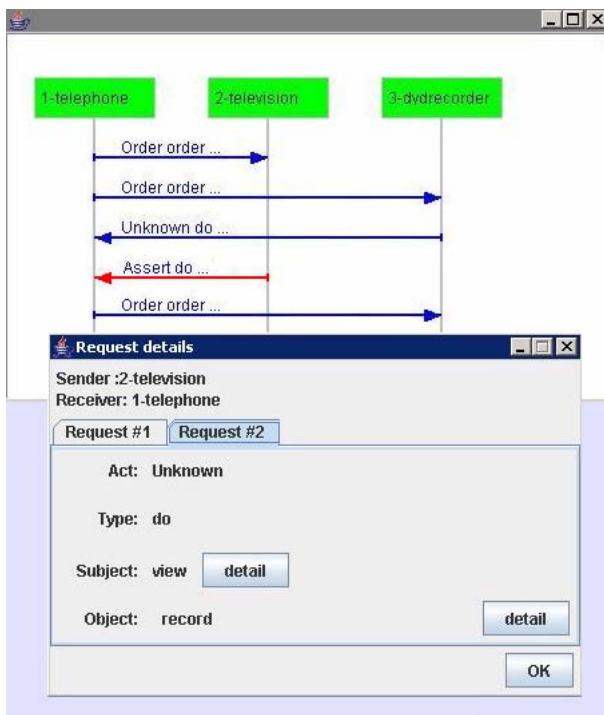


Figure 6 – A Screenshot of a VDLMessage Details.

6. CONCLUSION AND PERSPECTIVES

The aim of our work in the Ambient Intelligence research domain is to contribute to the development of devices communication. We outlined in this paper the importance and the usefulness of ubiquitous communication in a home environment for a human user. We proposed an approach in which home appliances are based on reasoning and interactive agents. We presented the underlying formalisms ensuring such properties. Then we described the interaction protocol allowing the agent-based appliances to dialogue and compose in order to achieve a task without having any knowledge about their pairs. Lastly, we demonstrated our algorithms behaviour in a home appliances scenario.

Though our protocol seems to offer roughly the features of the FIPA CNP protocol [FIPA-CNP] (8), it provides for agents the advantageous feature to compose and coordinate dynamically their capabilities in order to achieve a task. Moreover, using dialogues with high-level requests, our interaction protocol can be extended to support many other interaction scenarios.

Our interaction protocol gave good results in the presented scenario. However, it presents some shortcomings that we have to address in our future work to obtain a generic interaction protocol.

- Currently, our protocol delegates the agent initiating an interaction to broadcast unsolved messages, whereas we also would like to enable participant agents to do that, according to the interaction context (in our example, the DVD recorder could ask directly the telephone). In a general way, we would like to enable agents to select dynamically the protocol they will be involved into [Quenum and Aknine] (13).

- We would our protocol to support feedbacks to the user. Typically, we would like an agent to ask the user when it encounters situations with conflicts, multiple choices, when it can't perform an action du to a missing element, etc.

- Our agents should take other initiatives when they don't receive an acknowledgment sometime after sending a triggering order.

- We would like our agents to store information about the agents' capabilities, which could be provided during agents' interactions. This may allow agents to collect information about their environment and achieve a task in a pragmatic manner.

Therefore, we believe that AmI work could take significant advantages if it benefited from the Multi-Agent System (MAS) communities research, especially in agents communication, coordination and composition.

REFERENCES

1. Abascal J., Civit A., Fellbaum K., 2001, "Smart Homes - Technology for the Future", COST Seminar "Telecommunications: Access for all?", Leuven, Belgium.
2. Booch G., Rumbaugh J., and Jacobson I., 1996, "The Unified Modeling Language User Guide", Addison-Wesley, Reading, Massachusetts.
3. Cabac L., 2003, "Modeling Agent Interaction Protocols with AUML Diagrams and Petri Nets", Hamburg, Germany.
4. Ducatel K., Bogdanowicz M., Scapolo F., 2001, "Scenarios for Ambient Intelligence in 2010". European Commission Information Society Directorate-General.

5. Eggen B., Hollemans G., Van de Sluis R., 2002, "Exploring and Enhancing the Home Experience". Cognition, Technology and Work. Springer-Verlag.
6. FIPA. FIPA Interaction Protocol Library Specification, 2001.
<http://www.fipa.org/specs/fipa00025/XC00025E.pdf>
7. FIPA-ACL. The FIPA ACL Message Structure Specifications, 2002.
<http://www.fipa.org/specs/fipa00061/>
8. FIPA-CNP. FIPA Contract Net Interaction Protocol Specification, 2002.
<http://www.fipa.org/specs/fipa00029/index.html>
9. Gárate A., Herrasti N., López A., 2005, "GENIO: An Ambient Intelligence Application in Home Automation and Entertainment Environment", sOc-EUSAI, ACM Press, NY, USA.
10. Lashina, T.A., 2004, "Intelligent Bathroom", European Symposium on Ambient Intelligence, Eindhoven, Netherlands.
11. Sabouret N, 2002, "A Model of Requests about Actions for Active Components in the Semantic Web". STAIRS'02, 11–20.
12. Sabouret, N., 2003, "Active Semantic Web Services: A Programming Model for Agents in the Semantic Web", EUMAS'03.
13. Quenum J.G., Aknine S., 2005, "A Dynamic Joint Protocols Selection Method to Perform Collaborative Tasks", CEEMAS'05, 11–20.
14. Vallée M., Ramparany F., Vercouter, L., 2005, Ubimob'05, 120, 85-192