Dynamic Service Composition in Ambient Intelligence Environment

Yazid Benazzouz, Nicolas Sabouret
Informatics Laboratory of Paris 6, 104 avenue du Président Kennedy, 75016 Paris, France
benazzouz@emse.fr, nicolas.Sabouret@lip6.fr

Belkacem Chikhaoui
Computer Science Department, Faculty of Science, University of Sherbrooke, Sherbrooke QC, J1K 2R1 Canada
belkacem.chikhaoui@usherbrooke.ca

Abstract

In Ambient Intelligence (AmI) environments, some services provided by AmI devices are often not visible to users and to other devices. The existing approaches deal with services’ composition and discovery as two independent parts. In this paper, we propose an alternative approach based on logical reasoning agent system. This system is supported by a communication protocol where agents discover automatically services provided in their environment and construct dynamically composite services. The service composition is constructed from an exchange of idiomatic expressions among agents and users, while the discovery process takes the form of an information request via the communication protocol. The advantage of this approach is that agents are able to acquire knowledge from each other and when interacting with users. This capability will facilitate the satisfaction of user’s requirements in an intelligent way. This study shows that agents are able to satisfy new services previously unknown to the system.

1 Introduction

Ambient intelligence aims at making use of ubiquitous computing, ubiquitous communication and intelligent user interfaces in order to provide the user with an environment which offers services when and if needed by the user[8]. According to J. Lindenberg et al. the large heterogeneity and dynamic nature of users, services and environments lead to the problem of how to identify and activate the appropriate service [11]. Moreover, if there is no single service that can satisfy the functionality required by the user, there should be a program or a software agent which can combine existing services together in order to fulfill the request.

We quote two central approaches used in the AmI design system, the Service Oriented Computing(SOC) and Multi-Agents System (MAS). Service Oriented Computing has emerged as a design system for AmI given similarities with the Web domain. The main motivation is to reuse Web methods for service implementation and deployment in ambient environment. The major issues that confront the development of SOC in pervasive computing environments is the capability to automatically compose services at design time or even better at runtime.

On the other side, multi-agents paradigm fits well to AmI environments requirements from several perspectives. In fact, MAS provides decentralized control based on distributed autonomous entities supported by complex interactions. Such features seem essential in AmI environments dealing with various, heterogeneous information from physical sensors, services or users preferences. According to M. Vallee et al. MAS adoption in the pervasive computing and AmI fields is still limited [20]. They did not seek to exploit the full potential of MAS.

In this paper, we propose a dynamic service composition method using logic-based program synthesis, by combining two essential tools prolog and gorgias ¹. The overall system exploits the potential of the View Design Language (VDL) which is a multi-agents platform [19]. Logic theorem proving offers a practical approach for decision making and provides higher expressive power of modeling preferences policy. Our solution addresses a response for two main issues: how to define AmI services and how are they composed via logic-based program. Moreover, we consider services and their invocation methods unknown to other agents and also to users. We have two assumptions about agents:

- agents don’t have prior knowledge about services offered by other agents in the AmI environment. This assumption is admissible by the fact that we use a

¹www.cs.ucy.ac.cy/~nkd/gorgias
multi-agents system, where agents are not necessary informed about the knowledge of other agents.

- agents don’t have prior knowledge about the service requested by the user. Since a user can request a service in which tasks that compose it or some parts of them are not necessary provided by the agent itself.

The process of composition will have the possibility to dynamically bind services at runtime, which means that this process will take advantage of the change of services availabilities.

The rest of this paper is organized as follows. Section 2 begins the discussion on related work. In section 3 we present our approach of service composition. Section 4 studies the agents architecture. In section 5 we give the communication protocol model. Finally, we conclude by describing the prototype implementation and a conclusion.

2 Related work

Much research has been done in the area of service composition in the past few years especially in the web area. In this section we review and discuss some important methods for service matching and discovery, service invocation and service composition approaches. We don’t make distinction among web services and AmI services due to the fact that, both environments share similar concepts and implementation issues.

The matchmaking is responsible for selecting appropriate services to achieve a goal. This component has a set of (existing) services in input and a requester as goal; the output is a set of services that can fulfill the goal[6]. This process presents two essential difficulties, the capture of user specification and the selection of the appropriate service. Usual service selection methods such as hierarchical menus or lists of services are insufficient to help the user in locating the appropriate service(s)[16]. The services list must be actively maintained[21], and often[1] leads to poor discovery results, because the keywords in the query can be different semantically or syntactically from the terms in a service description.

The keywords of the service description are used in our approach by the formal composition and the communication protocols. We believe that using keywords don’t affect heavily this approach for one reason. In Aml environment, few words suffice to identify the service and using ontologies solution will make more complexity. Service discovery is a process of locating one or more documents, that describe a particular service[1]. The service languages that support the above definition are UDDI, WSDL and SOAP. Walton[21] proposed a protocol-based formalism, a syntax and semantics of a protocol language which express precisely how the interaction with a service should be performed. One challenge for discovery methods is to reduce configuration effort e.g. if a device is added to the environment, other devices must be able to discover and to use new proposed services.

Given a set of existing services, how a method can find a composition that satisfies the user requirements? One interesting approach is proposed by S. Narayanan and S. McIlraith[15], where OWL-S descriptions are mapped first into a situation calculus and then encoded into a Petri net formalism. Petri nets representation provides possibility of web service composition. McIlraith and Tran[14] proposed a way to adapt and extend the Golog language for automatic construction of web services composition. Golog is a logic programming language built on top of the situation calculus (a logical language for reasoning about action and change). Golog interpreter will populate its knowledge bases from ontologies of individual services, user preferences and so on. These ontologies will be compiled into situation calculus representation. However, OWL-S[7] has been receiving some critics as revealed from the literature review. One remarkable disadvantage in term of web service composition is the lack of variables, which limits possibilities to describe data flows in a service composition. Additionally, OWL-S is not endowed with necessary composition functionalities. Furthermore, several researchers[17, 14, 15] have found that such issues as runtime behavior of service composition, exception handling and verification of composition correctness are not covered sufficiently in the current specification of OWL-S.

In [12, 3], a composite service is modeled by a graph. [3] used EFlow which is a platform for the specification, enactment and management of composite services. The graph is created manually but it can be dynamically updated. The graph may include service, decision and event nodes. However, such setup implies heavy reconfiguration efforts in fact, the availability of services may change very frequently in a highly dynamic environment like AmI.

3 Our Service composition approach

In this section, we investigate how to represent service composition patterns using an argumentation-based framework[9, 10] with dynamic preferences.

3.1 General Idea

The general idea of our service composition method is as follows. Given a set of existing services, the method finds a composition that satisfies the user requirements. The construction of such composition is conditioned by the choice between existing services. A service is chosen to belong to
a composition, if it is supported by an admissible argument. An argument is admissible if it defends itself against any attack. The notion of attack [10] between arguments is based on the possible conflicts between a literal and its negation, and on the priority relation. Therefore, the description of existing services and the requirements for the composition are translated to logical axioms, and they are specified in form of a rules sequent to be proven. If the answer is positive then the next step is to invoke the service from the generated proof. The reason that we use argumentation-based decision making as service composition method is supported by the fact that:

- The notion of argumentation can be extended to the context information e.g. able or not able to offer a service or temporary disconnected from the system. Thus, the agent will construct arguments upon its context.
- The agent must be able to argue it’s service composition, which it will be a self justification sufficient to approve the process.
- The argumentation-based decision making can be used to make decision on the services’ selection and composition.
- Due to the soundness of the logic fragment, the correctness of composite services is guaranteed with respect of the initial specification.

The argumentation system in Gorgias is based on the computation of admissible arguments using the attack relationship between arguments. Starting from an initial argument, the computation of an admissible argument is done by adding suitable defences for the initial argument. The system repeats the whole process until there is no defence for an attack against the basic argument, or until there are no more attacks.

We take advantage of the existing Gorgias programming languages based on Prolog as the automated theorem prover in our experiments. Logic theorem proving offers a practical approach for decision making and provides higher expressive powers of modeling preferences policy.

3.2 Agent Composition Framework

The behavior of an agent making a composition is conditioned on three parts:

1. The first part captures agents background information about the services and providers (agents) in the AmI environment.
2. The second describes the different relationships between services in a composition.
3. The last one describes preferences policy between a set of services.

The overall decision of which composite service to select, is based on the integration of these theories by exploiting an argumentation-based reasoning.

3.2.1 Background Language

The background language \( \mathcal{L} \) contains a set of offers and services types of the form: \( P(X, S, [A]) \), where

- \( P \) is a performative type belonging to the set of performatives. We will assume that agents share the same set of performative. We use a set of \( P \) more suited to our purpose: \( \text{performatives} = \{\text{offer, rent, share, ask, choose}\} \)
- \( X \) is the provider of a service
- \( S \) is a basic service
- \( A \) is a set of attributes depending on the description of the proposed service and can be omitted.

Example

Suppose two agents, \( tv \) agent and \( phone \) agent which offer a display service in a house. The equivalent formulation is of the form: \( \text{offer}(tv, \text{display, high}), \text{offer}(phone, \text{display, low}) \) with high and low being the quality rating of their screens. Someone can ask for display e.g. \( \text{ask}(X, \text{display}) \). Suppose another agent \( P \), which can rent channel at some cost. Such service will be expressed by the from \( \text{rent}(p, \text{canal}+, 15) \). One other agent \( Q \) can share a game between many users e.g. \( \text{share}(q, \text{game, worldcup}) \) or a video e.g. \( \text{share}(q, \text{video, Spiderman3}) \).

3.2.2 Composition Language

The composition language \( \mathcal{CL} \) contains a set of basic relation restrictions associated to service and inspired from DOLCE formalism [2]. This set of basic primitive relations suitable to characterize our composition commitments as neutrally as possible. The composition types are of the form: \( R(S, S') \) where,

- \( S \) and \( S' \) are services.
- \( R \) is composition type belonging to the set of relations. We use a different set of relations more suited to our purpose \( \text{relation} = \{\text{partof, kindof, tpartof}\} \)
- \text{partof} defines a dependence between \( S \) and \( S' \), it asserts that, the service \( S \) must be always executed as a part of the global execution behavior of the service \( S' \).
- \text{kindof} defines a dependence between \( S \) and \( S' \), it asserts that, the execution of the service \( S' \) can be replaced for some reason by the execution of the service \( S \).
• `tpartof` defines a dependence between `S` and `S'`, it asserts that the service `S` will be temporarily executed as a part of the global execution behavior of the service `S'`. For example, `S_1` and `S_2` are two services such that `S_1` is kind of `S_2`. If `S_1` is used in the service composite `S_c`, and for some reason `S_1` is unavailable, then `S_2` can temporary replace `S_1`. We say that `S_2` is tpartof `S_c`.

**Example**

Suppose two services, switching off the house light and switching off only the bed room light. Thus, switching off the light of the bed room will be always executed when the user request to switch off all the light of the house. We express it by a rule of the form `tpartof(switchBedRoomLightOff, switchHouseLightOff)`.

3.2.3 Preference Language

Preference language `PL` contains a set of preferences between rules of the form: `prefer(r, r')`, means `r` is preferred than `r'` where `r` and `r'` are rules expressed in `L`, `CL` or `PL` languages. Thus, we can make preferences on preferences.

3.3 Personal Theory of an Agent

At this stage, using the above languages we can compose a private or personal theory of an agent that captures its own view of the services and their composition in four parts.

- `Basic component Tbackground`, that defines the private description of services and providers.
- The preference component, `Tpreference`, that defines the general preference policy.
- The composition component, `Tcomposition`, that captures private characteristics of a composition.
- The common knowledge of agents, `Tcommon`, that captures the common knowledge to all agents.

3.3.1 Basic Component

Basic component contains rules that are (for an agent `X`) of the form:

\[
\begin{align*}
  r_1 & : P(X,S,[A]) \leftarrow true & (1) \\
  r_2 & : P(X,S,[A]) \leftarrow C_j & (2)
\end{align*}
\]

Where `i` belongs to the label set of offer type, `j` belongs to the label set of possible requirement and `C_j` (which can be empty) is called requirement enabling conditions of the offered service. In other words, these are the conditions under which the agent `X` provides a service `S` with or not a requirement.

**Example**

An agent `X` asks for a service `S` in which its quality can be low or high. `r_1 : ask(X, S) \leftarrow offer(X, S, low)`

The Rules of basic component will be translated to Gorgias syntax as follows:

\[
\begin{align*}
  r_1 & : P(X,S,[A]) \leftarrow true \land \text{rule}(r_1, P(X,S,[A]), []) & (1) \\
  r_2 & : P(X,S,[A]) \leftarrow C_j \land \text{rule}(r_1, P(X,S,[A]), [C_j]) & (2)
\end{align*}
\]

3.3.2 Composition Component

Composition component contains rules and are (for an agent `X`) of the form:

\[
\begin{align*}
  r_1 & : R(S, S') \leftarrow true & (3) \\
  r_2 & : R(S, S') \leftarrow C_i & (4)
\end{align*}
\]

The first rule expresses the default composition, while the second rule states that under some conditions, the composition will be done.

**Example**

if we consider the composition example mentioned above, we will have a rule as follows:

\[
\begin{align*}
  r_1 & : P(X, S, [A]) \leftarrow true \land \text{composition}(r_1, P(X, S, [A]), []) & (1) \\
  r_2 & : P(X, S, [A]) \leftarrow C_j \land \text{composition}(r_1, P(X, S, [A]), [C_j]) & (2)
\end{align*}
\]

3.3.3 Preference Component

The rules in the preference component `Tpreference` express priorities over the composition step rules.

\[
\begin{align*}
  r_{hij} & : \text{prefer}(r_{ki}, r_{ji}) \leftarrow true & (5) \\
  r_{ijk} & : \text{prefer}(r_{ji}, r_{ki}) \leftarrow C_{jk} & (6)
\end{align*}
\]

where `C_{jk}` are specific conditions that are evaluated in the background knowledge. The first rule expresses the default preference of responding with `r_{ki}` over responding with `r_{ji}`, while the second rule states that under some conditions, the preference is the other way round.

**Example**

We consider two agents which offer the same service with different quality.

\[
\begin{align*}
  r_1 & : ask(X, S) \leftarrow offer(X, S, low) \\
  r_2 & : ask(X, S) \leftarrow offer(X, S, high) \\
  r_3 & : neg(ask(X, S)) \leftarrow true; \text{ i.e. neg()} \text{ express the negative of ask(X, S).} \\
  r_4 & : prefer(r_3, r_1) \leftarrow true \\
  r_5 & : offer(te, display, high) \leftarrow true \\
  r_6 & : offer(phone, display, low) \leftarrow true
\end{align*}
\]

In this example, the rule `r_3` will have always the high priority than `r_1`; which means that the service with high display quality will always be chosen.
3.3.4 Common Knowledge

The rules in the common knowledge $T_{common}$ make the relation between previous components.

$$r_{i} : call(X, Y, S') \leftarrow P(Y, S, [A]), [R(S, S')]$$ (7)

This rule expresses an argument for the agent X to consider the service offered by the agent Y. The agent Y must offer the service requested or offers a service intervening in the composition of the requested service.

**Example**

$$call(X, Y, switchHouseLightOff) \leftarrow offer(Y, switchBedRoomLightOff), partof(switchBedRoomLightOff, switchHouseLightOff)$$

i.e. the agent X has an argument to apply Y for providing the service switchBedRoomLightOff.

The Rules of basic common knowledge will be translated to Gorgias syntax as follows:

$$r_{i} : call(X, Y, S') \leftarrow P(Y, S, [A]), [R(S, S')]$$

$$\lor rule(r_{i}, call(X, Y, S'), [P(Y, S, [A]), [R(S, S')]])$$

4 Multi-Agents Architecture

We have designed and implemented the respective agents on an agent architecture VDL-platform[18, 19]. The VDL-platform is a multi-agents environment, which provides supports for cooperative work between the participants agents. The VDL model is based on XML tree rewriting. The agent’s description is an XML tree whose nodes represent either data or actions. The agent rewrites the tree at every execution step according to its specific elements. This model allows agents to access at runtime to the description of their actions, and to reason about it for planning, formal question answering and behavior recognition[13]. Agents can interact with each other by sending messages.

4.1 VDL Agent composition Model

The agent model described in figure 1 is made up of many components

1. UI User Interface, where user can express the service needed in XML or in natural language e.g. “save the movie”, this request may lead to an open dialog with the agent. To illustrate our approach we used sentences in the form [and] < verb > < noun >. The < noun > can be just a noun or a composite of < determiner > < noun >. The vocabulary set can be defined respecting the context domain.

2. VDL engine is the principal behavior of the agent, where all agent tasks are coordinated.

3. AIP Agent Interaction Protocol, it permits to the agent to engage agent-agent dialogs using three kinds of protocols, SD ( Services Discovery protocol), SI ( Service Invocation ) and LP ( Learning protocol).

4. KB knowledge base, necessary for agent reasoning. The KB is a set of logical rules.

5. Service Composition Select and compose services which satisfy the user specification. It is quite common that many services have the same or similar functionalities e.g. the service “display”. Agents with more services qualities and availability are further solicited.

The agent behavior algorithm consists in three main steps: wait for arrival of certain event, perform internal actions and/or interact with user and/or other agents, and add one or multiple knowledge to it’s knowledge base.

4.2 Description of Basic Services

Our basic services are described as indivisible tasks act toward increasing the the number of possible composition, thus the system will find more combined services. We use VDL[19] to describe basic services for three reasons:

- VDL language permits to define actions and data which can be combined in a closely manner to define a service.
- We are interested by VDL as a candidate to define services in AmI environment.
- VdlAgents have the capabilities to reason on their own code, and this is very interesting for the agent to self-reconfigured its code or its service description e.g. we can imagine an agent which have a data with some value. If the value seems to be erroneous in execution, the agent is able to detect such problem and correct it without initialization of the system.

A service has two parts. The first one, is related to its identification information which can be revealed to all agents in the environment. The second describes the service task to be executed. The first one is presented as VDL data and the second as a VDL action. Information about a service describe the name of the service, its quality information which can be ( low, medium or high) and one
or more parameters. The definition is open to other specific addition.

```xml
<service>
  <service-name/>
  <service-ident-value/>
  <service-param/>
  <service-quality/>

more description can follow here
</service>

An action will contain the necessary information for the execution process of one service. When the action is executed, it must verify that the corresponding service is authorized. This is defined in the `guard` tag which makes a comparison between the `enable` tag value (contains the service identification value of the service to provide at given moment) and the `service ident value` of the action.

```xml
<action>
  <name>service</name>
  <guard>
    <get><enable/></get>
    <value>(service identification value)</value>
  </guard>
  <task code>
</action>
```

### 4.3 Description of Composite Service

The service composition process is a deductive reasoning task, which transforms the user-agent interaction to a set of rules. These rules will describe the composite service. The agent has two answers “how?” and “that is all” to all user entries. The user responds with “ok” to finish the interaction with the agent.

**Example**

user: save the match
agent: how?
user: search in teletext
agent: that’s all
user: and record in vcr
agent: that’s all
user: ok

The equivalent rules to the above dialog are:

```xml
rule\{(f1, tpart(\{search, save\}), [])\}.
rule\{(f2, tpart(\{record, save\}), [])\}.
```

The composition in this way is a hard task because it is made with no prior knowledge about existing services. Thus, we adopt a service description close to the human understanding. However, we will be confronted to the keywords problem discussed previously.

### 5 Communication Model

Our agents communicate by sending messages, as described in [4]. Agent communication language `ACL` is described below: `{id, sender, receiver, content, in – reply – to}`, the content is described as `{request_i}_i\in N` and a request is a couple of `<α, γ>`, where α is the performative of the request, and γ is the object of the request. A performative can be:

1. `Rorder`, represents a request to another agent to perform an action e.g. display the calling number,
2. `Rhow`, used by the agent when it has to ask about a service composition e.g. “how to save the movie?”,
3. `Rassert_is`, used to send service composition e.g. saving a movie consists on searching in TV program for the start time and date and synchronize the storage with VCR,
4. `Rassert_can`, sending an assertion on its service’s capabilities e.g. i can search in channel program,
5. `Rack`, to confirm the accomplishment of a service execution e.g. i have finished saving the movie,
6. `Rwhat_can`, it’s a query about agents capabilities e.g. what can you do,
7. `Rassert_cannot`, used to refuse the execution of a service e.g. i’m busy now.

To make coordination between the SD, SI and LP protocols, we developed a coordination algorithm which permits to the agent to handle the upcoming messages. For this purpose, we developed an XML script for creating a Finite State Machine (FSM). Our script consists of a set of states, a set of transitions, a start state, a final state and variables. All these items can be listed in any order. FSMs are created by vdfsms compiler written in Java language and overlapped in theVDLObserver [18]. An FSM description is given as follows.

```xml
<fsm>
  <name>
  <initState>
  <finalState>
  <state>
  <transition>
  <from>
  <to>
  <event>
  <action>
</transition>
</fsm>
```

![Figure 2. SI protocol](image)

The FSM name will be used as the name of the Java object that represents this FSM. The states should be a list of XML tags and
Every time cycle {
    for each message in queue {
        if the message contains a valid Fsm event do
            if it exists a corresponding Fsm for the event then deliver it
            else {
                delete the corresponding Fsm model
                Update its state and number
                deliver the messages to it
            }
        else finds its corresponding action if it exists
        forward to the next message
    }
}

Figure 3. FSM Coordination algorithm

Figure 4. Our ambient environment example

the state name need to be unique. < initState > is the state that the FSM will begin in. It must be one of the states defined in the state list. < finalState > is the final state of the FSM. The heart of a FSM remains in its transitions. In our FSMs, a transition consists of four elements, an initial state, an event, an action and a final state. The initial state and the final state need to be one of the states defined in the state list. An event acts as a precondition for an action. Any message that arrives to the FSM Coordination component that is not addressed to specific FSM, activates a new FSM or it will be handled singly. In addition, any agent can create FSMs for its own events to lay out specific tasks. Figure 5 shows our FSM Coordination algorithm.

6 Implementation

The prototype is the main tool to verify that the proposed approach is an applicable solution. It also paves way for evaluating the approach in both qualitative and quantitative manners. This section is focused on functionality specification, as well as some technical details that are specialized for this system. We have used the prototype with an example illustrated by the figure 4.

Using his phone, a user requests his tv to save a football match because he will be away when the match is playing live. The aim of this example is to compose a service from two unitary services offered by two different agents (tv and vcr) in the environment. We will present the different features of the prototype.

First of all, the user interacts with the phone agent as shown in figure 6, and presented in the dialog below which describes the external language used in the interaction.

user: save the match
agent: how?
user: search in teletext
agent: that's all
user: and record in vcr
agent: that's all
user: ok

The external language is translated to the given rules and added to the Knowledge base of the agent.

\[
\text{rule}((1, \text{part}(\text{search}, \text{save})), [])
\]

\[
\text{rule}((2, \text{part}(\text{record}, \text{save})), [])
\]

The discovery process presented in figure 5 shows the interaction between agents. The phone requests capability of services from every agent in the environment (tv and vcr). The result of the discovery process and the translation from the VDL description of services to the Gorgias rules syntax, is detailed in the following rules.

\[
\text{rule}((11, \text{offer}((\text{tv}, \text{display}, \text{high})), []))
\]

\[
\text{rule}((12, \text{offer}((\text{tv}, \text{search})), []))
\]

\[
\text{rule}((13, \text{offer}((\text{tv}, \text{tvon})), []))
\]

\[
\text{rule}((14, \text{offer}((\text{tv}, \text{tvoff})), []))
\]

\[
\text{rule}((21, \text{offer}((\text{vcr}, \text{play})), []))
\]

\[
\text{rule}((22, \text{offer}((\text{vcr}, \text{record})), []))
\]

The given rule:

\[
\text{rule}((1, \text{offer}((\text{tv}, \text{display}, \text{high})), []))
\]

\[
\text{rule}((2, \text{offer}((\text{tv}, \text{search})), []))
\]

\[
\text{rule}((3, \text{offer}((\text{tv}, \text{tvon})), []))
\]

\[
\text{rule}((4, \text{offer}((\text{tv}, \text{tvoff})), []))
\]

\[
\text{rule}((5, \text{offer}((\text{vcr}, \text{play})), []))
\]

\[
\text{rule}((6, \text{offer}((\text{vcr}, \text{record})), []))
\]

The given rule:

\[
\text{rule}((r1(\text{X}, \text{Y}, \text{Q}), \text{callee}(\text{X}, \text{Y}, \text{S}), [\text{offer}(\text{Y}, \text{Q}), \text{tparse}(\text{Q}, \text{S}))])
\]

will be used in the prove process. An admissible argument Delta for the query prove(call(phone, Y, save), Delta) is Delta = [f1, f12, r1(phone, tv, save, search)] and Delta = [f2, f21, r1(phone, vcr, save, record)].
Therefore, the agent will use the invocation protocol to invoke sequentially the search service and the recorder service from tv and vcr respectively.

7 Conclusion and Perspectives

In this paper, we have elaborated the four different components forming the service composition system, namely, vdlms compiler developed with Java and the prover based on Prolog and Gorgias. The implementation is of prototype quality. We integrated those components through the VDL multi-agents system. The correctness of the generated result is guaranteed by the correctness of the forum programs. The quality of the result and the performance evaluation are not very important at this stage, we may test the prototype with more complex knowledge and composition features. This tests will permit the improvement of the composition system.

The benefit of this approach is to study the service composition and discovery from the viewpoint of logic-based program and communication protocol. We argue that argumentation-based decision making combined with communication protocol offers a practical approach to the success of service composition. The main contributions of this paper are summarized as follows. First, a generic framework for the purpose of presenting an abstract protocol based on FSM model is developed for VDL platform, and finally specific system based on the generic framework has been developed.

Context awareness is becoming an essential feature of services targeted for ambient intelligence. Our future work will be essentially focused on this problematic.

References


