SOA-based Service Layering for Facilitating Dynamic Service Composition and Alternative Service Discovery∗

MEEYEON LEE, JUNG-WON LEE**,†, SEUNG SOO PARK AND WE DUKE CHO**

Department of Computer Science and Engineering
Ewha Womans University
Seoul, 120-750 Korea

**Department of Electrical and Computer Engineering
Ajou University
Suwon, 443-749 Korea

SOA (Service-Oriented Architecture) is a methodology that is gaining popularity as a system design paradigm. In this paper, we propose metaservice as a SOA-based service abstraction for ubiquitous computing. The key to service modeling of SOA-based systems is in classifying services into business, application, and orchestration layers. However, service modeling techniques for SOA cannot be directly applied to a ubiquitous computing environment, which is dynamic, autonomous, and largely affected by run-time context. We have mainly focused on establishing the service layer to allow users to specify the computing goal using abstract description, detailed description, or both. The layered service abstraction also supports dynamic service composition and alternative service discovery to achieve the goal. Experiments show that the proposed metaservice provides a 49%-81% improvement in completing the execution of services over composition using none-layered services and a 30%-47% reduction in binding time by facilitating alternative service discovery.

Keywords: service-oriented architecture, ubiquitous computing, community computing, metaservice, service ontology, service modeling, dynamic service composition

1. INTRODUCTION

Ubiquitous computing space is a dynamically coordinated and connected set of heterogeneous and possibly mobile computing entities. In a ubiquitous computing environment, the user’s requested services may be realized depending on various aspects related to the run-time context such as request time, the user’s location and preference, service availability, and so on. Service-Oriented Architecture (SOA) is an effective architecture for managing ubiquitous computing space. This architectural paradigm promotes the development of software systems as sets of collaborative and loosely coupled services [1, 2]. Unlike existing component-based software development approaches, SOA facilitates composition and assembly of services regardless of communication protocols, enabling enterprise-level system integration of B2B as well as systems within an enterprise. The core principles of SOA are based on reusability, autonomy, loose-coupling, abstraction, and composability.
SOA recommends that the developer and software architect perform service-oriented analysis and design in order to comply with service-oriented principles. The last step of the service-oriented analysis is for service modeling, wherein the service and operation candidates are extracted from a decomposed business process and the service interface layer is classified according to the role of a service [3]. In Fig. 1, the service interface layer is positioned between the application and business layers. It is classified into three layers, the application service for interfacing with the application logic, the business service for handling business rules, and the orchestration service for collaboration. This hierarchical layer is an important basis of service modeling and promotes the development of software that can respond with flexibility and speed to ever-changing customer’s demands and market opportunities [4]. However, the service modeling technique of SOA cannot be directly applied to a ubiquitous computing environment due to the following reasons.

First, unlike the business process of the upper layer, the workflow for ubiquitous computing is not predefined. If we define a workflow in advance, we consider only the static composition of services. Currently, most researches on workflow for ubiquitous computing have largely focused on dynamic service composition [5, 6]. Essentially, there are two types of service composition mechanisms, ‘static’ and ‘dynamic’ [7]. In static composition, the elements that would go into the composed service are known a priori, whereas in dynamic composition the required individual services are located on demand using a service discovery method. Therefore, with modeling approaches based on a predefined workflow, we cannot consider dynamic service composition with context awareness.

Second, the goal of a business service tends to be clear but that of ubiquitous computing may be very complex, ambiguous, and diverse. For example, the billing service of B2B consists of sub-steps such as invoice submission, order fulfillment, notification process, and so on. There are common business rules for the billing service and the goal is achieved by executing sub-services. However, in a ubiquitous computing environment, the execution of services for attaining the goal depends on the user’s preference and request. Namely, users or developers may describe computing goals using different abstraction levels of services. For example, in a ubiquitous computing space, Fig. 2 below can be a possible scenario for “making an environment suitable for sleeping”.

Fig. 1. Service layer of SOA [3].
Ms. Smith wants the lamp turned off and the curtain closed when she falls asleep. Therefore, the required services, ‘Turn off the light (S1)’ and ‘Close the curtain (S2)’, are invoked for service composition. This is not likely to be dynamic in terms of the scenario. However, the service is dynamically bound with the available devices according to where Ms. Smith is sleeping. In the case of Mr. Wilson, he requests only help to sleep, but his personalized sleeping condition is used as an input. Therefore, the profile of Mr. Wilson is analyzed and S1, S7, and S21 are called. His goal description appears to be simple and abstract but multiple predefined services are called to make the sleep mode. Lastly, Mr. Johnson specifies detailed conditions such as 30 lux for illumination, 20°C for temperature, and so on. In this case, we do not know what services are composed during description of the computing goal, as they depend on the run-time context. If the current temperature is under 20°C, ‘Turn on heater’ or ‘Turn on boiler’ will be necessary whereas if it is above 20°C, ‘Turn on cooler’ should be invoked. Ms. Smith, Mr. Wilson, and Mr. Johnson have the same goal for setting the sleep mode but describe the goal with different abstraction levels.

Accordingly, a technique to allow various abstraction levels to describe the computing goal as well as dynamic composition is needed. In this paper, we mainly focus on establishing the service layer to allow users to specify the goal using abstract description, detailed description, or both. The service layer also supports dynamic service composition and alternative service discovery for achieving the goal. Experiments show that the proposed metaservice provides a 49%-81% improvement in completing the execution of services for achieving the goal over composition using non-layered services and a 30%-47% reduction in binding time by facilitating alternative service discovery.

The remainder of this paper is organized as follows. Section 2 compares related work and section 3 explains previous work for collaboration models and the definition of metaservice for ubiquitous computing. Section 4 describes the process for service modeling and the specifications of metaservice for discovering and composing services that are context-aware of ubiquitous environmental parameters. Section 5 proposes the hierarchy of the metaservice library by service overloading and presents the mechanism of dynamic service composition and alternative service discovery using a service overloading method. Section 6 presents a preliminary application and results and section 7 concludes the paper.
2. RELATED WORK

Research on workflow for ubiquitous computing has mainly focused on dynamic service composition. In PICO [6], all services and user tasks in a pervasive computing environment are described as directed graphs and submitted to the directory. In the directory, the components of the task graph are matched with available basic service graphs during runtime. If a direct match does not exist, PICO makes a complex service in order to improve the service availability. The complex service is a combination of available services and can provide the same function to meet the given task. It is formed by comparing input and output parameters of basic services on the service graph and then finding the path from the matched input to the output. However, the composition scheme has a limitation in that it does not consider other effects that occur in the path, because it only deals with inputs and outputs of the user-task graph. That is, a complex service may include services as internal nodes that a user does not request, and as such it may actually be an obstacle to the achievement of the goal.

Aura [8] reconfigures an environment for the user task by finding suppliers that can seamlessly provide specified services regardless of environment changes such as the user’s location and availability of resources. More specifically, Aura calculates and selects the best combination of suppliers that can provide the required services, satisfy the user’s preference, and maximize the quality of each service in a given environment. However, they do not propose a service structure to connect semantically or manage services, suppliers, and resources.

In Gaia [9, 10] an application is composed of five components: adapter, model, controller, presentation, and coordinator. The ‘application adaptation’ mechanism of Gaia supports user tasks to execute continuously regardless of changes such as spatial movement and device failure. However, most previous works including Gaia, DIANE [11], and ICrafter [12] do not clearly distinguish the concept of a service from a function of a device. They focus on ‘functions of devices’ as the target of composition and availability. However, services that are used to support ubiquitous computing range widely from simple services such as ‘turning off a boiler’ to valuable or high-level ones such as ‘making an environment comfortable to read something’, ‘therapy service for anti-stress’, and so on.

On the other hand, SOA recommends modeling a service that is valuable in itself. Making services valuable is an important basis of reusability, loosely coupling, and composability. For this, four information types were proposed for SOA – technology, functional, programmatic logic, and quality of service – as ‘types of meta abstraction’ [13]. However, they are formal types for publishing services and interfacing with them. For ubiquitous computing, it is necessary to design a semantic service abstraction that can take into account the runtime context.

To describe semantics, ontology can be employed. In [14], an ontology-driven web services composition was proposed. The compositions are obtained by checking semantic similarities between interfaces of individual services. The parameters in the interfaces are expressed as objects such as ‘wine’, ‘beverage’, and ‘seafood’ and use a directed graph similar to PICO. The relationship between objects is appropriate for building a domain ontology. However, for layering services into business, application, and orchestration layers, it is necessary to build a service ontology rather than a domain ontology.

With respect to QoS, [15] takes user utility values or service quality factors into
account for web service selection and composition. The service quality factors are execution price, duration, reputation, reliability, and availability, which produce the optimal web services using graph-based selection algorithms [15]. However, the service-oriented principles of SOA focus on reusability, composability, autonomy, and loose-coupling. Therefore, we need to form the service layer with consideration of service-oriented principles.

3. SERVICE-ORIENTED ANALYSIS FOR UBIQUITOUS COMPUTING

Currently, in the area of ubiquitous computing, the concept of a service is close to the function of a device, because service-oriented analysis is often overlooked when building services. In SOA, service modeling through a service-oriented analysis is the core process. Therefore, we propose an ontology-based service layer for ubiquitous computing services.

3.1 Community: Metaphor for Describing a Complex Goal

In our previous work [16], we defined the concept of ‘community’ for modeling a ubiquitous computing environment. Fig. 3 explains the concept.

In a ubiquitous smart space, after the situation from a sensor is recognized (step 1), the goal is set (step 2), and the appropriate community is created (step 3). Community members (devices or human) then cast and collaborate with each other (step 4) to achieve the goal.

Here, we have used a template-based approach for composing a community and designed CDL (Community Description Language) [17] to describe the community template. The major part of CDL is describing the ‘role’ of a member and the collaboration rule. Most existing researches in this aspect focus solely on binding fixed roles with various members during dynamic service composition. In other words, they predefine what kinds of services are needed to attain the goal and then a service discoverer selects only available devices according to the run-time context. However, in the real world, we cannot exactly know which service, e.g. ‘turn on the light’ or ‘turn down the light’, is needed to achieve the goal, ‘make an environment for a living room comfortable to read a book’. For example, in the case of a human community for a music concert, we define
only roles of the staff such as the producer, stage designers, sound technicians, and stagehands but do not fix the roles with services such as ‘turn on’, ‘turn off’, or ‘turn up’ in detail before running. It is inefficient or may be impossible to describe detailed roles in a community template. Therefore, we should design a service abstraction to enable description of abstract roles of a service.

3.2 Metaservice: Service Abstraction

In this paper, we define ‘metaservice’ as an abstracted and structured representation of a service. Fig. 4 shows the relative position of metaservice among role, member, and services.

Fig. 4. Metaservice.

We classify the view as ‘community view’ and ‘implementation view’ and re-classify it according to the aspect of the binding time – the ‘conceptual’ service is needed when we write a community template and the ‘real’ service during the community execution. For example, a user can use ‘manageTemperature()’ metaservice to specify the role for controlling temperature in a community template; then boiler becomes a member of the community and executes a concrete service ‘Turn on($boiler)’ if the run-time is winter and it need to raise the temperature of the user’s location.

We define the conceptual service for describing the role as metaservice and the concrete service as a real service name for binding during the execution. In a business domain (see Fig. 1), the business rule establishes the difference between the application logic and the business process, whereas here, the important clue to distinguish the conceptual abstract layer from the concrete service layer is the effect of a service. We can find a method to encapsulate services by examining the environmental or functional effect after executing a service. Fig. 5 shows our proposed service layer and metaservices.

As shown in Fig. 5, we defined 7 environmental or functional effects. The abstract-level services are categorized by the effects but they cannot execute operations by themselves. On the contrary, the atomic services correspond to the device operations and they can execute the specific operation of the device. Other remaining services are defined as composite-services between an abstract and an atomic service.

- Effect: the environmental or functional effect after executing a service.
Abstract service: a service of the highest level. It can be grouped according to the environmental/functional effect and consists of one or more atomic services, composite services or both.

Composite service: a service of middle levels. It consists of one or more atomic services, other composite services or both.

Atomic service: a service of the lowest level. It cannot be partitioned any further, i.e., it cannot have children.

4. SERVICE MODELING

The top-down approach of SOA is an effective methodology for service modeling. It is an “analysis first” approach and based on ontology. In this section, we propose a process for modeling metaservice.

4.1 Extracting Metaservice Candidates from Scenarios

We use a collaboration diagram of UML to extract service candidates from text scenarios. Objects and relations can be easily represented in collaboration diagrams. In a diagram, each device or entity is represented as a class object, each operation or function is represented as an object link, and its name is given as a link message.

Fig. 6 (a) shows a collaboration diagram that describes the second scene of the scenario for maintaining the sleep mode in a home environment. From the diagram, we can obtain the object names, their attributes, and service names by simply listing them and inferring hidden ones. For example, we can acquire object names such as ‘Curtain’, ‘Light’, and ‘Lamp’; user names such as ‘wife’ and ‘husband’; and service names such as ‘LightOn’ and ‘CurtainClose’ in the ‘Home’ scenario. We may infer other entities and services such as ‘LightOff’ and ‘CurtainOpen’. Fig. 6 (b) shows a part of the domain ontology, built by OWL [19], for a home. Objects such as devices or users are defined as classes and their relations or attributes are defined as properties. The domain ontology only explains the relationships between objects whereas we should focus on operations of objects for forming the service layer. Therefore, we extract operation names from link messages in the diagram and rename them as service names.
4.2 Metaservice Specification

Metaservice is the concept to abstract and effectively structure operations of devices that can exist in a ubiquitous environment and it is used to describe roles of communities. For building a service ontology, we define the specifications for metaservice as given in Table 1.

- **SID**: service identifier.
- **ServiceName**: a service name consists of words capitalized and contains no spaces.

<table>
<thead>
<tr>
<th>SID</th>
<th>ServiceName</th>
<th>SceneNo</th>
<th>AbsLevel</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s$:ScenarioNumber,ServiceNumber</td>
<td>string</td>
<td>integer</td>
<td>$[0, 1, 2]$</td>
</tr>
<tr>
<td>Child</td>
<td>Object</td>
<td>TerminatingService</td>
<td>none</td>
</tr>
<tr>
<td>$\text{opt}(SID_i, SID_j, \ldots, SID_n)$</td>
<td>none</td>
<td>${obj_1, obj_2, \ldots, obj_n}$</td>
<td>none $\parallel SID_k$</td>
</tr>
</tbody>
</table>

**Precondition**

<table>
<thead>
<tr>
<th>Status</th>
<th>Location</th>
<th>Priority</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>$obj_i.status$</td>
<td>$?location$</td>
<td>$?priority$</td>
<td>${i_1, i_2, \ldots, i_n}$</td>
</tr>
</tbody>
</table>

**Effect**

<table>
<thead>
<tr>
<th>Environmental/Functional</th>
<th>Device</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Effect Parameter$</td>
<td>$obj_i.status \parallel value$</td>
<td>$N/A$</td>
</tr>
</tbody>
</table>
However, if the first word of a service name is a verb, it should start with a lowercase letter.

- **SceneNo**: the related scenario number.
- **AbsLevel**: three abstraction levels of services. ‘AbsLevel = 0’ indicates an atomic service, ‘1’ means a composite service, and ‘2’ is an abstract service.
- **Child**: in the case of a composite or an abstract service, its child services are specified as opt(SID_i, SID_j, ..., SID_n). Here, ‘opt’ includes operations such as Choice, AnyOrder, and Sequence.
- **Object**: a platform or smart objects that can operate the service. For example, the service ‘musicOn’ has ‘MP3’, ‘Audio’, ‘TV’, ‘PC’, and so on.
- **TerminatingService**: the service that terminates the executing service. (Currently, we assume that the ‘TerminatingService’ is the service that can make a device turn-off.) However, in the case of Abstract services, we specify ‘none’, because it is ambiguous to determine the terminating condition.
- **Precondition**: four types of preconditions for executing the service. The service, for a given task, is selected by ‘Precondition’ and ‘Effect’. The ‘Status’ of the object is checked before executing the service. For objk.status, ‘Status’ is one of {on, off, open, close}. For example, in the case of ‘LightUp’ service, the specific illuminator has to be turned on before calling it. Binding information such as ‘isLocatedIn(?person, ?location)’ is important for selecting the most appropriate device to execute the service. ‘Priority’ is the priority among services with the same effect and is referred when selecting the most appropriate service. It is determined according to the user preference, energy efficiency, environmental context (like season), and so on. Variables such as ‘?location’ and ‘?priority’ are obtained from the context broker. ‘Input’ means parameters necessary to execute the service.
- **Effect**: Types of effects that can occur after executing a service. ‘Environmental/Functional’ refers to effects such as illumination, temperature, noise, display, anti-stress, *etc*. ‘Device’ presents the object status after executing the service. ‘Output’ is the result value.

### 4.3 Service Ontology

Based on the specifications described in the previous section, we build a service ontology using OWL-S [20]. This is an OWL variation that was developed to facilitate building service ontology. Unlike OWL, which is appropriate for ontology in a declarative style, OWL-S is aimed at more process/procedure-oriented service ontologies. To represent our specifications, we adopted and selected the subset of OWL-S features, because OWL-S focuses on web service description and includes many web service-specific features such as service grounding. Table 2 shows selected OWL-S features and mapping between our specifications and OWL-S.

In OWL-S, detailed operations of web services are represented as ‘processes’ and a ‘process model (process ontology)’ is employed to describe the properties of each process. We can treat our metaservice as a process in OWL-S and describe ‘Object’, ‘Precondition’ and ‘Effect’ of our specifications by using ‘hasInput’, ‘hasOutput’, ‘hasPrecondition’, ‘hasResult’ properties, which are employed to describe IOPE (In put/Output/ Precondition/Effect) of OWL-S specifications. We represent the super-sub relations
Table 2. OWL-S features to represent our service specifications.

<table>
<thead>
<tr>
<th>Our Service Specifications</th>
<th>OWL-S</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract/Composite service (‘ServiceName’ &amp; ‘AbsLevel’)</td>
<td>Composite process</td>
</tr>
<tr>
<td>Atomic service (‘ServiceName’ &amp; ‘AbsLevel’)</td>
<td>Atomic process</td>
</tr>
<tr>
<td>‘AbsLevel’ &amp; ‘Child’</td>
<td>Control constructs</td>
</tr>
<tr>
<td>‘Input’ (of ‘Precondition’)</td>
<td>hasInput property</td>
</tr>
<tr>
<td>‘Status’ &amp; ‘Location’ (of ‘Precondition’) &amp; ‘Object’</td>
<td>hasPrecondition property</td>
</tr>
<tr>
<td>‘Environmental/Functional’ (of ‘Effect’)</td>
<td>hasResult property</td>
</tr>
<tr>
<td>‘Output’ (of ‘Effect’)</td>
<td>hasOutput property</td>
</tr>
<tr>
<td>The rest</td>
<td>Comment</td>
</tr>
</tbody>
</table>

of metaservices by using ‘control constructs’ such as ‘Sequence’, ‘AnyOrder’ and ‘Choice’ and the rest of our specifications with ‘comment’. ‘AbsLevel’ and ‘Child’ in our service specifications (described in Table 1), the main features of service structuring, are represented by the process model of services.

5. SERVICE COMPOSITION USING METASERVICE

The concept of metaservice and service ontology proposed in this paper are used to support the context-aware and dynamic service composition, which is the main role of three components for community computing – community manager, context broker, service discoverer. The following describes the roles of each component.

- Community manager: controls the creation, organization, execution, and termination of a community. It can request context information to a context broker and service discovery to a service discoverer. It is responsible for achieving a community goal by binding the services specified in a community template to the actual services discovered by the service discoverer.
- Context broker: manages context information such as sensing data, user data and events in ubiquitous environments. It infers some useful information from context in run-time, and sends them to service discoverer and community manager.
- Service discoverer: finds the most appropriate service by using service ontology and context information from context broker, when abstract or composite metaservices are specified in a community template, or it is necessary to discovery alternative services. And it informs the selected service to a community manager.

In the following sections, we explain methods to select and provide the best service for users with our service ontology.

5.1 Metaservice Overloading

In this section, we present a method for metaservice overloading, which makes it possible to invoke the most appropriate service from metaservices at the execution time. Originally, ‘method overloading’ was one of the features of various OOP languages that
allows the creation of several functions with the same name that differ from each other in terms of the type of input and output of the function [18].

Definition Service Signature

Service Signature of S, \( \sigma(S) = [(p_1, p_2, \ldots, p_n) | \text{none}, E] \), where \( p_k \) is the kth input parameter and \( E \) is a required effect. The types of input parameters can be the status of an object and input values, which are specified by users or obtained from a context broker in the run-time. \( E \) is obtained from a context broker that collects and analyzes contexts in the run-time.

For example, service signature of a service ‘LightUp’ is defined as \( \sigma(\text{LightUp}) = [\text{ illuminanceValue}, E-\text{IlluminanceUp}] \). In Fig. 7, service signatures are shown in the gray boxes attached to services. Metaservice Overloading is a method of finding the most appropriate service among services with the same name in different contexts. If a user specifies an abstract or composite metaservice as a role at the design time, the most appropriate service is invoked according to the ‘service signature’ in the run-time. \( E \) is obtained from a context broker that collects and analyzes contexts in the run-time.

For example, service signature of a service ‘LightUp’ is defined as \( \sigma(\text{LightUp}) = [\text{ illuminanceValue}, E-\text{IlluminanceUp}] \). In Fig. 7, service signatures are shown in the gray boxes attached to services. Metaservice Overloading is a method of finding the most appropriate service among services with the same name in different contexts. If a user specifies an abstract or composite metaservice as a role at the design time, the most appropriate service is invoked according to the ‘service signature’ in the run-time. For example, assume that three services have the same name, ‘S’, in different contexts i, j, k with signatures \( \sigma(S_i) = [(p_1, p_2), E_i] \), \( \sigma(S_j) = [(p_1, p_2, p_3), E_j] \), and \( \sigma(S_k) = [(p_3), E_k] \). Suppose we use the service name ‘S’ at the design time. If a set of input contexts for achieving the goal includes \( p_1, p_2, \) and \( p_3 \) and the result of service execution derives the effect, \( E_2, S_j \) will be invoked. Here, \( S_j \) is considered to be the most appropriate service among \( S_i, S_j, \) and \( S_k \).

Fig. 8 shows the algorithm for metaservice overloading. In this method, we compare service signatures of all services that have the same name with the service specified by the user \((\odot)\) in terms of the number \((\Box)\) and types of input parameters \((\circ)\) and required effect \((\square)\). If a service satisfies three conditions \((\Box \sim \square)\), it is selected as one of the service candidates \((\bigcirc)\). After the for-loop, if there is just one candidate, then it becomes...
Fig. 8. Algorithm for metaservice overloading and alternative service discovery.

Table 3. Metaservice overloading (in the case of ‘manageLight’).

```
<table>
<thead>
<tr>
<th>σ</th>
<th>P1</th>
<th>P2</th>
<th>E</th>
<th>Service Binding</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>—</td>
<td>—</td>
<td>E-IlluminanceUP</td>
<td>LightOn(E-IlluminanceUP)</td>
</tr>
<tr>
<td></td>
<td>—</td>
<td>E-IlluminanceDown</td>
<td>LightOff(E-IlluminanceDown)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>?illuminanceValue</td>
<td>—</td>
<td>E-Illuminance</td>
<td>LightControl(?illuminanceValue, E-Illuminance)</td>
</tr>
<tr>
<td></td>
<td>?sign</td>
<td>?illuminanceValue</td>
<td>E-Illuminance</td>
<td>LightControl(?sign, ?illuminanceValue, E-Illuminance)</td>
</tr>
<tr>
<td></td>
<td>on</td>
<td>?illuminanceValue</td>
<td>E-Illuminance</td>
<td>LightUp('on', ?illuminanceValue, E-Illuminance)</td>
</tr>
<tr>
<td></td>
<td>on</td>
<td>?illuminanceValue</td>
<td>E-IlluminanceDown</td>
<td>LightDown('on', ?illuminanceValue, E-IlluminanceDown)</td>
</tr>
</tbody>
</table>
```

the most appropriate service (⑤). However, if there are multiple candidates, the one with the highest (?priority) is selected (⑦).

In Fig. 7, the abstract service, ‘manageIlluminance’, encapsulates 12 services from atomic to composite services. Using the concept of metaservice overloading, we can bind the appropriate service dynamically in the case of ‘manageLight’ service, as shown in Table 3. When ‘manageLight’ service is bound to ‘LightControl’ service, the metaservice overloading method is used recursively.

For example, if we call ‘manageLight()’, ‘LightOn’ or ‘LightOff’ may be invoked because it has no input parameter (rows 1 and 2 in Table 3). At that time, if the required effect (E) is set as E-IlluminanceUp, ‘LightOn’ will be executed (row 1). In the case of ‘manageLight()’ with run-time context ‘+30 lux’, LightControl(+, 30, E-Illuminance) is selected (row 4), and then it is interpreted as E is ‘E-IlluminanceUP’ because of the input value, ‘+’ and ‘30’. Hence, service ‘LightUp(30)’ is executed (row 5).

We can select the best service among 12 possible services based on the run-time context by using the metaservice overloading method even though a user describes a role
with the abstract service ‘manageLight’. It is very efficient to allow a user to describe an abstract goal. The abstract description, e.g. ‘manageIlluminance’, can increase service reusability and availability, because it has a hidden layer (shown in Fig. 7). The following sections demonstrate the effectiveness of the service layer using metaservice overloading.

5.2 Diversity of User’s Goal Description

The following describes three scenes for the same goal, ‘making the environment suitable for sleeping’.

- Scene 1 (using abstract-level services)
  A user is sleeping. Call just makeComfortableSleepEnv().

- Scene 2 (using composite-level services)
  A user is sleeping. If noise occurs, call ‘VolumeControl’. If the temperature or the illumination is not comfortable for sleeping, call ‘manageTemperature’ or ‘LightControl’.

- Scene 3 (using atomic-level services)
  When a user tries to sleep, the temperature of the bedroom is increased (by HeaterOn) and the light is turned off (LightOff). If his/her sleeping conditions are disturbed by the temperature falling below 23°C and the noise exceeding 30dB, call ‘VolumeDown’ and ‘HeaterOn’. If the user is awakened from sleep, the light is turned on (LightOn).

5.3 Dynamic Service Composition

We explain how metaservices in the previous three scenes are bound to concrete services dynamically based on the run-time context. Fig. 9 shows the part of our metaservice ontology, which is composed of metaservices of three types. The ‘Users’ can describe their goals by using metaservices in various levels. The ‘Service Discoverer’ discovers and binds appropriate services according to the run-time context in the case of abstract goal descriptions by using metaservice overloading.

For the first scene, if we assume that makeComfortableSleepEnv() for ‘Scene 1’ consists of three abstract services, ‘manageIlluminance’, ‘manageTemperature’ and ‘manageSound’, all services may be candidates (in Fig. 9 (a)).

In Fig. 9 (b), a user specifies three metaservices, VolumeControl(), manageTemperature(), and LightControl(), but the service discoverer selects concrete services among many sub-services of the specified services based on the run-time context. The service, ‘manageTemperature’, may be finally bound to ‘HeaterOn’, because the required effect is E-:TemperatureUp that is derived by the context broker. Even though we do not know exactly which service is needed when we define the community template, the concrete service of ‘manageTemperature’ can be found in hidden layers of the service.

However, the case of the last scene is different from the first and second. Four metaservices, HeaterOn, LightOff, VolumeDown, and LightOn, are selected and each service is bound to available objects such as a boiler for HeaterOn, a bed lamp for LightOn, and so on according to the user’s location or preference. Rather than considering which service has to be selected, device availability is considered, because the aforementioned are atomic services. In Fig. 9 (c), four atomic services are bound and the available objects are show in a blue-colored box.
(a) Binding abstract services (for scene 1).

(b) Binding composite services (for scene 2).

(c) Binding atomic services (for scene 3).

Fig. 9. Dynamic service composition and binding.
5.4 Alternative Service Discovery

A mechanism for alternative service discovery is needed when the operating device is out of order, when the requested service is not available, and when a more efficient service or device is newly entering into the space. Fundamentally, our metaservice ontology structures and groups all metaservices by environmental or functional effects. Therefore, when specified services are not available, alternative services that have the same effect can be discovered and replace unavailable ones by using the structure of our metaservice ontology. In other words, among other services that have the same ancestor as the specified service, the one with the same effect can be an alternative service. Metaservice overloading algorithm in Fig. 8 is also used to find alternatives. We maintain a candidate list in a tree-like priority queue, and whenever alternative service is required we can pick the best service from the priority queue (○8). In Fig. 7, if a user specifies ‘LightDown()’ metaservice to lower the illumination, but the metaservice is unavailable in the run-time, the system will find ‘LightOff()’, ‘CurtainClose()’, ‘CurtainCloser()’ metaservices, which have the same effect, i.e., ‘E-IlluminanceDOWN’, as ‘LightDown()’, among sub-metaservices of ‘manageIlluminance()’ metaservice, which is the root of all services related to illumination control. The most suitable metaservice will be selected from among these 3 metaservices according to the run-time context, such as user preference and whether an operable device exists. Here, in order to enhance the degree of user satisfaction, we assume that it is preferable to replace services with alternatives that can provide the most similar effect rather than fail in community creation.

6. IMPLEMENTATION AND EXPERIMENTAL RESULTS

We have built a total of 165 metaservices to implement ‘well-being apartment life’ and ‘public safety’, which are composed of 13 abstract services, 42 composite services, and 110 atomic services.

6.1 U-Home Service

We have built a service ontology for ‘well-being apartment life’ and tested three community templates for making a comfortable environment for sleeping, as described in section 5.2. Fig. 10 shows the details of the U-Home service.

When the wife is preparing to go to sleep, the community for sleeping (Sleeping Service) is created. The services, ‘manageLight’, ‘manageTemperature’, ‘MusicTherapy’ and ‘NoiseControl’, are invoked. If all requested services are available, the air conditioner regulates the temperature (for ‘manageTemperature’), the bedroom light dims, the curtains are closed (for ‘manageLight’), and atmospheric music plays (for ‘MusicTherapy’). Soon, her husband comes home late and turns on the TV in the living room. At this time, ‘TV service’ is created and ‘VideoOn’, ‘LightControl’, and ‘manageCurtain’ start running. However, his wife tosses and turns in bed because of the TV sound (noise over the specific dB) and the living room light entering under the door. At this time, ‘NoiseControl’ and ‘manageLight’ operate and adjust the TV volume and the living room light. However, the husband will likely not be satisfied with this situation. This
problem is related to the community policy that is used by a community manager to solve
the role conflicts between communities, or by a context broker to define the priority of
services such as energy efficiency, user preference, device location, and so on [21].

6.2 Experimental Results

It is necessary to test the efficiency of the service composition using our service ab-
straction. We randomly selected $n$ metaservices based on our ontology, and made a group
of the services for defining a community. For example, we selected 8 services randomly
and assigned 6 services to the 7th community such as $C_7 = \{S_1, S_2, S_{10}, S_{23}, S_{31}, S_{55}\}$ and 2
services to the 10th community such as $C_{10} = \{S_2, S_{67}\}$.

Our experiments have two purposes. The first is to show the degree of improve-
ment the metaservice provides in completing the community execution over composition
that considers only non-layered services. The second is to measure the time reduction in
binding services according to the service availability.

6.2.1 The success rate of community execution

A community is composed of multiple services. Even if a single required service is
not available among them, the community execution will fail without our alternative ser-
dvice discovery method – for example, when the device for providing the service does not
exist in the specific space, it is out of order, or it plays a role of another service. Namely,
the success rate of the community execution is largely affected by the service availability.
In this experiment, we repeated the unit experiment 1000 times and used the following
parameters.

- We select 61 atomic services from U-Home library. If all services are available, we set
  the availability as 100% in the space. We then subdivide the availability as into six
degrees – between 50%-100%.
- We generate 11 communities according to the abstraction level. Each community also
  has 10 variations, which consist of $n$ services, such as $n = \{6, 12, \ldots, 54, 61\}$. There-
  fore, we have 110 communities.
In Fig. 11, the X-axis is the ratio between atomic services and other services (abstract and composite) and the Y-axis indicates the success rate of community execution. The ‘70% ATOM’ refers to a community that has 17 abstract and composite services, and 37 atomic if \( n = 54 \). If the availability is 100%, all communities can be successfully executed. However, with availability between 50%-90%, as more atomic services are used, the success rate becomes accordingly lower. The use of atomic services beyond 50% brings on a rapid reduction of the success rate, to under 20%, regardless of the availability.

![Fig. 11. The success rate of service execution.](image)

In contrast, if we use abstract and composite services over 80% (indicated as 20% ATOM), the rate is increased sharply. On average, our proposed layer provides a 49%-81% improvements in completing the execution over non-layered services.

### 6.2.2 Service binding time

If the service is not available, we can wait to execute the community until it is available rather than fail. This is accomplished by discovering alternative services that have the same effect. Fig. 12 (a) shows how much time is required for binding meta-service with concrete services, using the same dataset and the availability of the previous experiment. We assigned 1 unit time to the binding time when a service is available and randomly assigned 1-10 unit time to the waiting (holding) time when a service is not available.

The X-axis is also the ratio between atomic services and other services and the Y-axis indicates service binding time. Service binding time is defined as the total time to discover alternative services plus to wait until the services are available. In the case of ‘100% ATOM’, the binding time is 33.1 when the services are 100% available and 107.7 when ‘50% available’ – the waiting time can increase drastically depending on the availability of services, because the system cannot find an alternative service and has to wait until the service is available. As we use more abstract services, the time is decreased – this is due to the structure of our service ontology, which makes it possible to find alternative services, instead of matching the specified services statically and postponing the execution until they become available to obtain a specific effect.
In Fig. 12 (b), we normalized the time basis as ‘time = 100’ at 100% ATOM for comparison. The results show that availability under 70% does not significantly affect the binding time. Therefore, it is concluded that the total time can be effectively reduced by using service abstraction.

7. CONCLUSION

We proposed the service layer for ubiquitous computing. It allows diverse user description levels as well as dynamic service composition and alternative service discovery. We show that the proposed service layer provides a 49%-81% improvement in completing the execution of services over composition using non-layered services and a 30%-47% reduction in binding time by facilitating alternative service discovery. Further research should be carried out to extend the domain to an ‘office’ environment, with a focus on computing power and mobility. It is also necessary to evaluate the degree of user satisfaction when the alternatives are executed.

REFERENCES


Meeeyeon Lee received a B.Sc. degree and M.Sc. degree in Computer Science and Engineering from Ewha Womans University in 2003 and 2005, respectively. She is currently a Ph.D. candidate in the Department of Computer Science and Engineering of Ewha Womans University. Her research interests include ubiquitous computing, AI, and ontology.

Jung-Won Lee received her Ph.D. in Computer Science and Engineering from Ewha Womans University, Seoul, Korea, in 2003. She was a Researcher of LG Electronics and did an internship in the IBM Almaden Research Center, U.S.A. Currently, she is an Assistant professor of the Department of Electrical and Computer Engineering of Ajou University, Korea. Her areas of research include SOA and ubiquitous computing.

Seung Soo Park received his Ph.D. in Computer Science from the University of Texas in Austin, U.S.A., in 1988. He is a Professor of the Department of Computer Science and Engineering of Ewha Womans University, Korea. His main research interests are ubiquitous computing, semantic web, AI, and data mining.

We Duke Cho received his Ph.D. in Electrical Engineering from KAIST, Korea, in 1987. He is the president of the Ubiquitous Computing and Network (UCN) Research Foundation, Director of the Center of Excellence for Ubiquitous System, and a Professor of the Department of Electrical and Computer Engineering of Ajou University, Korea. His main research interest is ubiquitous computing and intelligent systems.