Activity Policy-based Service Discovery for Pervasive Computing

Woohyun Kim[1], Saehoon Kang[1], Younghee Lee[1], Dongman Lee[1], and Inyoung Ko[1]

[1] Information and Communications University
119, Munjiro, Yuseong-gu, Daejeon 305-732, Korea
{woorung, kang, yhlee, dlee, iko}@icu.ac.kr

Abstract. Service discovery is an essential technique to provide applications with appropriate resources. However, user mobility and heterogeneous environments make the discovery of appropriate resources difficult. The execution environments will be rapidly changed, so developers cannot predict available resources exactly in design time. Thus, service discovery frameworks for pervasive computing must guarantee transparent development environments to application developers. In this paper, we introduce how to semantically describe and discover a variety of services in different environments. This approach helps applications to find appropriate services even though the required ones are not available or not found. For this, we propose a fine-grained effect ontology which is used to reasonably evaluate functional similarity among different services, and a policy-based query coordination which is used to dynamically apply different resource constraints according to human activities. Finally, we show with a feasible scenario how to find appropriate services in different environments. Our approach helps applications to seamlessly perform their tasks across a dynamic range of environments.

1 Introduction

Pervasive computing environments are surrounded by a variety of computing devices, software services, and information sources. Applications must adapt to dynamically changing environments to seamlessly perform their tasks [1, 2]. However, different environments have different resources, and their capabilities are also different. Even though developers specify in design time all the resources required by applications, appropriate resources may not always be available across a dynamic range of environments [1, 8]. Suppose that there are some real instances such as "AlarmClock," "TV," "Audio," and "Light" in a home environment. A developer would like to use an instance "AlarmClock" to wake a user out of a deep sleep. Unfortunately, if the instance was not available or not found in real time, which instances could be found and used instead of the "AlarmClock"?

Traditional service discovery systems (i.e., Jini, UPnP, SLP, and Salutation) are not proper for pervasive computing since syntactic matching is mainly used [3, 4]. They do not support alternative representations of semantically similar services. In contrast, context-aware service discovery systems [3-7] take advantage of contextual

information (i.e. location) syntactically or semantically to provide appropriate services in pervasive computing environments. Similarly, semantic service discovery systems [1, 2, 7, 8, 9] make use of abstract representations in various aspects of services to transparently provide appropriate services despite different environments. Nevertheless, these approaches do not seem to deal with yet systematically the semantics in various aspects of services: data semantics (requirement), functional semantics (capability), QoS semantics (effect), and execution semantics (execution pattern) [15]. In this point, Semantic Web [16] which semantically describes and discovers Web services in the IOPE (Input, Output, Precondition, Effect) level gives us a noble idea to reasonably provide substitutable services which are functionally similar to the required ones when they are not available or not found.

In the earlier "AlarmClock" example, some sophisticated inference processes are required to semantically evaluate functional similarity among other available candidates such as "TV," "Audio," and "Light." Note that "AlarmClock" has a sound effect, and "TV" and "Audio" also have another type of sound effects. In this paper, we define an *effect* as a process to change the state of the world to other state. Each sound effect has its own properties as follows: impact factor (*alarm volume*, *TV volume*, and *audio volume*), behavior pattern (*sound delivery*, *multimedia delivery*, and *music delivery*), and human perceptibility (*auditory sense*). The enumerated properties are constructed in some semantic hierarchies. That is, *TV volume* and *audio volume* have equivalent semantics to *alarm volume* with intensity beyond a certain level. Similarly, *multimedia delivery* and *music delivery* have equivalent semantics in the aspect of *sound delivery*. These semantic associations play an important role in evaluating which service can substitute the originally required service.

In this paper, we propose a fine-grained effect ontology which is used to reasonably evaluate functional similarity among different services, and a policy-based query coordination which is used to dynamically apply different resource constraints according to human activities. The goal of our approach aims at enabling applications to take full advantage of local resources across a dynamic range of environments.

The remainder of this paper is as follows. In Section 2, we describe existing efforts on service discovery for pervasive computing. Section 3 covers design considerations, and we introduce our approach in detail in Section 4. The implementation details of the proposed scheme are addressed in Section 5. Finally, we describe our conclusions and suggest future work in Section 6.

2 Related Works

Several research efforts have been done on a high level of abstract representations to solve the problems of mobility and heterogeneity. Gaia [1] introduces an application as a set of structural components , MPACC(model, presentation, adaptor, controller, and coordinator). To make applications polymorphic, such a component is represented as abstract functionalities in ontologies. The actual components are dynamically bound in a given environment. And yet, Gaia does not mention how systematically the ontologies are constructed. The ontologies are predefined by rule of thumb and made by hand. Aura [2] introduces a service as one of abstract semantics

of coarse-grained functionalities required to perform a user task. It proposes an idea to be automatically able to bring up all the resources associated with a given task. It works on a higher level of abstractions such as tasks as coalitions of abstract services. In other words, Aura does not tell what properties of a service are equivalent to those of its similar service(s).

On the other hand, several approaches use some policies to find appropriate resources. Olympus [10] proposes the separation of class and instance discovery to allow alternative services to be found, which is based on the functionality required by an original service. When resolving virtual entities into actual ones, it considers developer-specific constraints, space-level policies, class-/instance-level context-awareness, and utility function. Olympus insists that even entities of classes that are highly different from the class specified by the developer can be discovered and used. In this aspect, Olympus is very similar to the proposed scheme. CARISMA [22] uses application-specific policies to enable mobile applications to behave according to contexts such as bandwidth, CPU performance, and even other applications' behaviors. CARMEN [23] uses declarative management policies for migration, binding, access control, user preferences, device capabilities, and service component characteristics. However, these approaches do not consider that the semantics of exact or abstract representations might be dynamically changed in different situations.

3 Design Considerations

Service discovery for pervasive computing has to deal with a user's mobility and environment heterogeneity. This leads to the issue of how to formally describe a diversity of services and transparently discover appropriate services despite different environments. Thus, we consider two key issues in this paper: *transparent accessibility* and *high availability*.

3.1 Transparent Accessibility

Figure 1 illustrates some processes to discover the *Alarm* service in pervasive computing environments. In the developer's point of view, the *Alarm* service keeps an abstract representation level. However, in the system's point of view, the real semantics of the required *Alarm* service can be dynamically changed as *SoundAlarm*, *VibrationAlarm*, or *DisplayAlarm* according to users' current location. It means that the *Alarm* service can be bound with some real instances in the current execution environments which have sound effects, vibration effect, or display effect. The binding to real instances is determined in real time, so the *Alarm* service is transparently kept in abstract level. In conclusion, service discovery should work on abstract representations to describe and discover all of the services in pervasive computing environments.

3.2 High Availability

Figure 1 shows a need to find substitutable services when the best matching instance "AlarmClock" is not available or not found in a given environment. Suppose that a user sleeps late in the morning. An application should find the *Alarm* service to wake him up. But there is no available instance "AlarmClock." Which instance can be used instead of the "AlarmClock"?

In the user's point of view, a certain function such as *StartAlarming* of "AlarmClock" is supposed to be somewhat equivalent to other functions such as *TVTurnOn* function of "TV," *AudioPlayCD* function of "Audio," or even *LightTurnOn* function of "Light." To take full advantage of local resources, consequently, service discovery frameworks for pervasive computing must provide applications with other substitutable instances which are functionally similar to the best matching instance "AlarmClock."

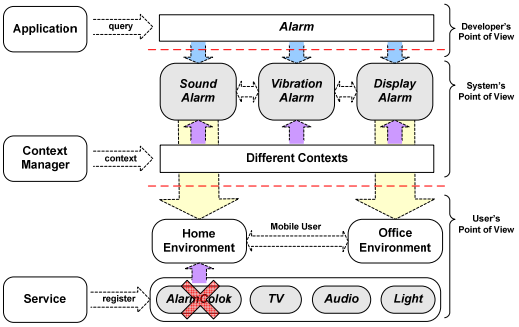

Fig. 1. The Discovery of the *Alarm* Service in Pervasive Computing Environments

4 Proposed Approach

Task-based computing [11, 12] and activity-based computing [8] focus on a higher level of abstraction level such as *tasks* or *activities* to enable computing environments to be aware of users' intents or requirements. In this paper, we especially pay our attentions to a high level of context model, *human activity* which means anything that users intend to do in a specific region by using some resources. The model is for representing what a user intends to do, what resources can be utilized for an activity, and where services can be performed. Thus, we use the *activity policy* which describes different resource constraints according to locations, humans, and activities. Furthermore, we use the IOPE model of Semantic Web [16] to describe the capabilities of services in a fine-grained level, and provide upper ontologies to reasonably evaluate functional similarity among different services.

4.1 Activity Policy-based Query Coordination

To describe different resource constraints according to human activities, Figure 2 shows how to define activity policies and how the user query is coordinated with the appropriate resource constraints in service discovery processes.

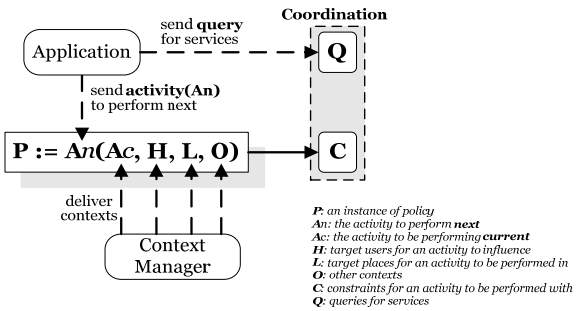

Fig. 2. Activity Policy and Query Coordination

At first, some policies, the notation "P := A*n*(A*c*, H, L, O)" is described. An application sends queries (Q) for services and the name of the next activity(A*n*). Our service discovery framework receives the contextual information such as the name of the current activity (A*c*), target user (H) for the activity to influence, target place (L) for the activity to be performed in, and other contexts (O). In this point, an appropriate policy is selected, and then some resource constraints (C) are extracted according to the current contexts. The given queries and the selected resource constraints are adjusted in the query coordination part. Thus, the proposed scheme can reflect the dynamic changes of real semantics between abstraction (queries) and context-awareness (policies).

Suppose that Alice should receive business-related messages. Location sensors recognize her current position, and deliver the sensed values of (x, y, z) coordinates or logical/physical space names to ContextManager. Here, ContextManager [21] is a part of context-aware middleware to aggregate contexts and interpret their semantics. Authentication sensors could also deliver the ID number of Alice to ContextManager. If ContextManager could recognize even human activities, our service discovery framework would infer appropriate resource constraints according to Alice's activities as Figure 3. In this point, the context model of Gaia [14] based on first order logic would be useful for such a context fusion.

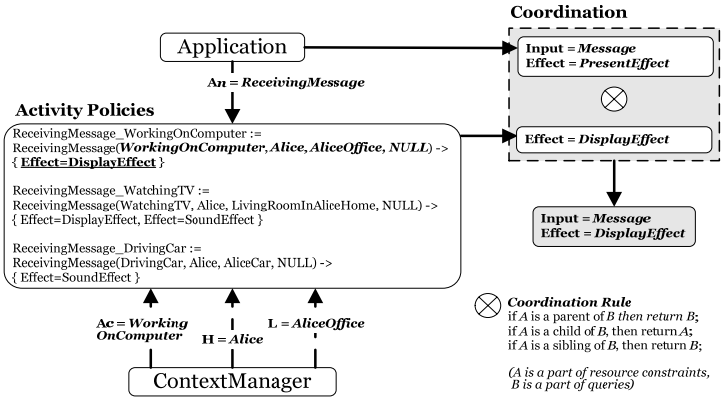

Fig. 3. An Example of Activity Policy-based Query Coordination

The application intends to find some local resources to present the messages to Alice as the query, Input=*Message* and Effect=*PresentEffect*. Our service discovery framework knows that Alice works with a computer at the office and the application wants to deliver some messages to her. Thus, the framework selects an appropriate resource constraint, Effect=*DisplayEffect*, and start coordinating it with the given query, Input=*Message* and Effect=*PresentEffect*. In this point, the effect parts between the query and the constraint trigger a conflict in the coordination part. To resolve this conflict, simple coordination rules are used with some subsumption relations such as the *parent*, the *child*, and the *sibling* relation in our ontologies as following.

- If *A* is a parent of *B*, then we select *B*;
- If *A* is a child of *B*, then we select *A*;
- If *A* is a sibling of *B*, then we select *B*;
  (*A is a part of resource constraints, B is a part of queries*)

Finally, the new coordinated query, Input=*Message* and Effect=*DisplayEffect*, enables Alice to receive the messages through "WindowsPopupMessage," when Alice works at the office not to distract other co-workers.

4.2 IOPE-based Semantic Matching

To maximize high availability of services in heterogeneous environments, it is very important to define how to semantically describe and discover services by using ontologies [15]. In this paper, we use the IOPE model to formally describe the capabilities of services in a fine-grained level. This enables services to be richly represented in data semantics, functional semantics, and QoS semantics.

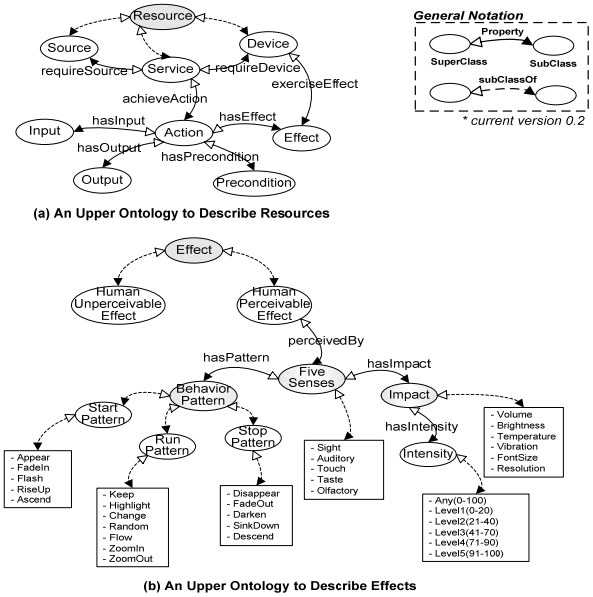

Fig. 4. Upper Ontologies to Formally Describe Various Resources

Figure 4 shows two upper ontologies designed by OWL [18] to describe the diversity of services and effects in pervasive computing environments. Ovals represent classes (or concepts), black arrows represent characterized properties (or roles), white arrows represent inverse properties, and dotted lines represent subsumption (parent and child) relations among some classes. In Figure 4 (b), the rectangle means some value instances of the parent class. That is, the parent class *Intensity* can be instantiated to one of the instance values such as *Any*, *Level1*, *Level2*, *Level3*, *Level4*, and *Level5*. Specific intensities can be defined in this way. In case of the volume of "TV" or "Audio," as an instance of the class *Impact*, a specific instance *SmallVolume* can be defined with the instance *Level1* as the value of the property *hasIntensity*. Similarly, another specific instance *LargeVolume* can be defined with the instance *Level5* as the value of the *hasIntensity*. In case of the brightness of "Light," some specific instances such as *WeakBrightness*, *ModerateBrightness*, and *StrongBrightness* can be respectively defined with some value instances of the *Intensity*. Therefore, we can make use of such a specific instance to evaluate functional similarity among some services.

At first, we define *effect* as a process to change the states of the world to other states. According to this definition, we consider three main properties: *perceivedBy*, *hasImpact*, and *hasPattern*. We understand some effects of services in human-centric aspects, so the effects which we deal with in this paper are those which can be perceived by the five senses of human beings. Moreover, the effects give some impacts on the states of the world, and those kinds of changes are achieved in the specific behavior patterns. Figure 4 (b) illustrate how to define the effects with the classes such as *FiveSenses*, *Impact*, and *BehaviorPattern*. To more easily explain, the following BNF format is used.

<*HumanPerceivableEffect*> := <*perceivedBy*> | <*perceivedBy*> <*op*> <*HumanPerceivableEffect*>
<*perceivedBy*> := <*FiveSenses*> (<*hasImpact*>, <*hasPattern*>)
<*op*> := and | or
<*FiveSenses*> := Sight | Auditory | Touch | Olfactory | Taste | …
<*hasImpact*> := Volume | Brightness | Temperature | Vibration | FontSize | Color
| Message | Music | Document | Multimedia | Image | …
<*hasPattern*> := Appear | Disappear | Increase | Decrease | Maximize | Minimize | FadeIn
| FadeOut | ZoomIn | ZoomOut | Deliver | Reserve | KeepInLimitTime | …

For example, the effect of "TVVolumeUP" action is to increase the volume of TV to a certain level. The effect is perceived by the auditory sense of human beings. Its impact factor is the volume of TV, and its behavior pattern is to increase. That is, the effect can be described as *TVVolumeIncreaseEffect* := *Auditory*(*TVVolume*, *Increase*). Similarly, the effect of "AudioVolumeUP" action is to increase the volume of Audio, so it can be described as *AudioVolumeIncreaseEffect* := *Auditory*(*AudioVolume*, *Increase*). In this way, we define a diversity of effects, and the effects are associated with some semantic hierarchies. Consequently, the effect of "TVVolumeUP" is similar to the effect of "AudioVolumeUP." Thus, both of them have the effect to make the volume of sound increased in a specific region. It is quite an interesting feature in pervasive computing environments since it shows that the different functions, "TVVolumeUP" and "AudioVolumeUP," can trigger some equivalent effects in a specific region. Such a feasible example is shown in Figure 5.

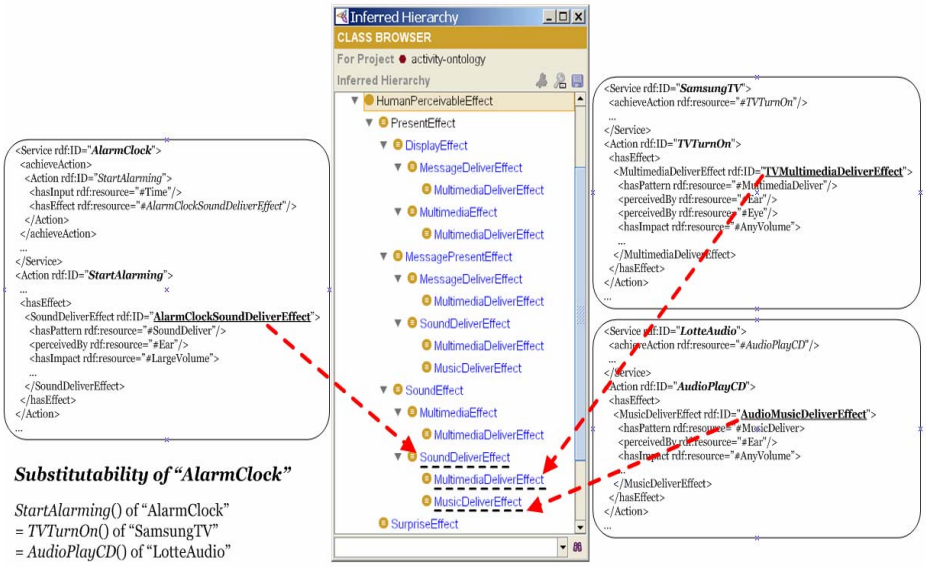

Fig. 5. Substitutability of "AlarmClock," "SamsungTV," "LotteAudio" in the Effect Ontology

When Alice sleeps late in the morning, an intelligent application checks her schedule and recognizes that she has an important business meeting today. Suppose that the application tries to find "AlarmClock" in the bedroom, but fails to find it. In this case,

it knows that some substitutable services such as “LotteAudio” or “SamsungTV” can be used instead of “AlarmClock.” To make it possible, “AlarmClock,” “LotteAudio,” and “SamsungTV” are described as Figure 5. “AlarmClock” service has a function “StartAlarming.” The function gives the effect *AlarmClockSoundDeliverEffect* on a target user. Similarly, “AudioPlayCD” function of “LotteAudio” has *AudioMusicDeliverEffect*, and “TVTurnOn” function of “SamsungTV” has *TVMultimediaDeliverEffect*. All the effects can be perceived by the auditory sense of human beings. Moreover, they have the behavior patterns to deliver some impact factors with different levels of intensities. Through the common properties, we have an inferred hierarchy in the effect ontology as shown in Figure 5. *AlarmClockSoundDeliverEffect* is an instance of *SoundDeliverEffect*, and *MultimediaDeliverEffect* and *MusicDeliverEffect* are inferred as subclasses of the *SoundDeliverEffect*. Each subclass has the instance *TVMultimediaDeliverEffect* and *AudioMusicDeliverEffect*, respectively. In conclusion, we can see that the different effects have a semantic hierarchy in the effect ontology by dynamically inferring the relative relations of the properties such as *FiveSenses*, *Impact*, and *BehaviorPattern*.

In the earlier part, we have already shown how to define some specific semantics related to *Intensity* and *Impact*. In Figure 5, each service was described with these specific instances. Initially, *AlarmClockSoundDeliverEffect* would be different from *AudioMusicDeliverEffect* and *TVMultimediaDeliverEffect* in the aspect of *Intensity*. Later, *AudioMusicDeliverEffect* and *TVMultimediaDeliverEffect* would have volume intensities beyond a certain level, which might be corresponded to that of *AlarmClockSoundDeliverEffect*. Through these processes, we can finally make sense that the effects of “TVTurnOn” function and “AudioPlayCD” function can be provided instead of that of “StartAlarming” function.

5 Implementation

We implemented the proposed scheme as the service discovery part in Active Surroundings [21], which is a middleware for pervasive computing environments. Figure 6 shows home appliances, location sensors, and IR (Infrared Rays) transceiver deployed in our prototype smart home environment. Especially, the IR transceiver is used to convey an IR signal corresponding to a user command to a target device.

Fig. 6. Home Appliances, Location Sensors, and IR Transceiver in the Prototyped Home

Each agent program corresponding to the devices is extended from the basic class ServiceProxy which has the fundamental interfaces such as discovery, registry, and invocation. The agent automatically registers its description file to the proposed service discovery server, which uses soft-state mechanism to maintain the state informa-

tion of each service. Our communications are implemented on HTTP protocols. Therefore, end-users can use local services in familiar ways. In addition, each service works as a server to allow other applications to invoke its service in peer-to-peer ways. Our framework manages all of the services in our ontologies, which are developed by using OWL [18] and Protégé [19]. Activity policies will be also managed in the ontologies, and inferred by Racer [20].

5.1 Activity Policy-based Service Discovery Framework

Figure 7 shows a hierarchical architecture of the proposed service discovery framework. ServiceProxy includes some operations to register, discover, and invoke services. Communications between ServiceProxy and ServiceDiscoveryServer are achieved on HTTP protocols. That is, the operations such as WWWRegister, WWWDiscover, WWWShow, and WWWCall are implemented as GET or POST methods of HTTP servers. ServiceProxy hides complex protocols required for the operations. Applications just use the given functions such as register(), discover(), and invoke(). In addition, general users can use the proposed scheme by using Web browsers such as Internet Explorer because we provide the operations as GET or POST methods of HTTP protocols.

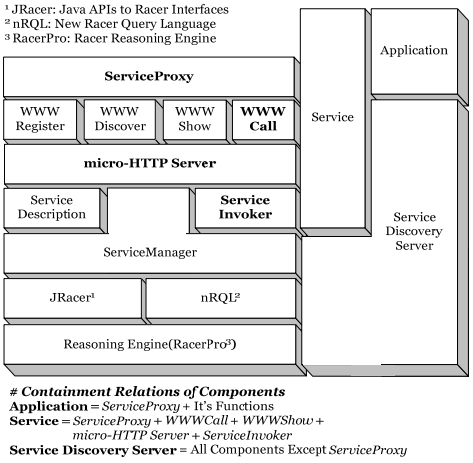

Fig. 7. Activity Policy-based Service Discovery Framework

On the other hand, all of the services must be extended from ServiceProxy to periodically register the service description files to ServiceDiscoveryServer. When the register() function is executed in each service, ServiceProxy creates a thread to automatically register the specific description file. The thread sleeps and wakes with the given lifetime which is described in the description file. The descriptions are stored in description repository and ontologies. Furthermore, ServiceProxy provides functionalities of micro-HTTP server to the services to support remote invocations. When applications invoke some functions in remote services, ServiceDiscoveryServer provides the reference to the desired services such as IP address, port number, class

name, and method name, and then the ServiceProxy in the side of applications invokes the remote calls to the micro-HTTP server of the target services. ServiceManager governs most of the operations which are occurred in ServiceDiscoveryServer. It closely interacts with the reasoning engine, RacerPro [20]. Activity policies and ontologies are operated with the reasoning engine. ServiceManager uses JRacer and nRQL [17] to perform policy-based query coordination and IOPE-based semantic matching.

5.2 Message Delivery Example

To illustrate the effectiveness of the proposed scheme, we use the activity ReceivingMessage introduced in Figure 3. We compare two types of service discovery approaches. Type 1 does not use policies, but use IOPE-based queries, while Type 2 uses policies with IOPE-based queries.

Table 1. Experiment Results of the Activity Policy, ReceivingMessage

	Application (Query)	ContextManager (Context)	PolicyManager (Constraints)	QueryPolicyCoordinator (Coordinated Query)	Service Discovery (Found Services)
Type 1 (Office)	discover("Message", null, null, "PresentEffect");	(WorkingOnComputer, Alice, AliceOffice)	N/A	(retrieve (?service ?action) (and (?input |Message|) (?effect |**PresentEffect**|) (?action ?input |hasInput|) (?action ?effect |hasEffect|) (?service ?action |achieveAction|) (?service |AliceOffice| |locatedIn|)))	(((?SERVICE |**VoiceSpeech**|) (?ACTION |SpeakMessage|)) ((?SERVICE |**WindowsPopupMessage**|) (?ACTION |PopupMessageDialog|)))
Type 2 (Office)	discover("Message", null, null, "PresentEffect", "**ReceivingMessage**");	(WorkingOnComputer, Alice, AliceOffice)	(?effect |**DisplayEffect**|)	(retrieve (?service ?action) (and (?input |Message|) (?effect |**DisplayEffect**|) (?action ?input |hasInput|) (?action ?effect |hasEffect|) (?service ?action |achieveAction|) (?service |AliceOffice| |locatedIn|)))	(((?SERVICE |**WindowsPopupMessage**|) (?ACTION |PopupMessageDialog|)))
Type 1 (Car)	discover("Message", null, null, "PresentEffect");	(DrivingCar, Alice, AliceCar)	N/A	(retrieve (?service ?action) (and (?input |Message|) (?effect |**PresentEffect**|) (?action ?input |hasInput|) (?action ?effect |hasEffect|) (?service ?action |achieveAction|) (?service |AliceCar| |locatedIn|)))	(((?SERVICE |**NateCarNavigation**|) (?ACTION |NateTTS|)) ((?SERVICE |**NateCarNavigation**|) (?ACTION |NatePopupMessageBox|)))
Type 2 (Car)	discover("Message", null, null, "PresentEffect", "**ReceivingMessage**");	(DrivingCar, Alice, AliceCar)	(?effect |**SoundEffect**|)	(retrieve (?service ?action) (and (?input |Message|) (?effect |**SoundEffect**|) (?action ?input |hasInput|) (?action ?effect |hasEffect|) (?service ?action |achieveAction|) (?service |AliceCar| |locatedIn|)))	(((?SERVICE |**NateCarNavigation**|) (?ACTION |NateTTS|)))
Type 1 (Home)	discover("Message", null, null, "PresentEffect");	(WatchingTV, Alice, LivingRoomInAliceHome)	N/A	(retrieve (?service ?action) (and (?input |Message|) (?effect |**PresentEffect**|) (?action ?input |hasInput|) (?action ?effect |hasEffect|) (?service ?action |achieveAction|) (?location |Home|) (?service ?location |locatedIn|)))	(((?SERVICE |**FreeTTS**|) (?ACTION |FreeTTSVoiceSpeak|)) ((?SERVICE |**CaptionMaker**|) (?ACTION |ShowSubtitle|)))
Type 2 (Home)	discover("Message", null, null, "PresentEffect", "**ReceivingMessage**");	(WatchingTV, Alice, LivingRoomInAliceHome)	(or (?effect |**DisplayEffect**|) (?effect |**SoundEffect**|))	(retrieve (?service ?action) (and (?input |Message|) (or (?effect |**DisplayEffect**|) (?effect |**SoundEffect**|)) (?action ?input |hasInput|) (?action ?effect |hasEffect|) (?service ?action |achieveAction|) (?location |Home|) (?service ?location |locatedIn|)))	(((?SERVICE |**CaptionMaker**|) (?ACTION |ShowSubtitle|)) ((?SERVICE |**FreeTTS**|) (?ACTION |FreeTTSVoiceSpeak|)))

Table 1 present the experiment results of the ReceivingMessage activity. Note that the queries are not changed according to different environments in the developer's aspect. However, in the system's aspect, the queries are dynamically coordinated in the QueryPolicyCoordinator component. When working on a computer in the office, Alice would like to receive message through "WindowsPopupMessage" not to distract other people. When driving a car, she wants to use "NateTTS" function for the sake of safety. In result, Type 1 might provide inappropriate services, while Type 2

provides appropriate services according to context aware resource constraints. Furthermore, our approach can take full advantage of local resources although mobile users move from one to another place. In this case, Alice can use a variety of local resources such as VoiceSpeech, WindowsPopupMessage, NateCarNavigation, FreeTTS, and CaptionMaker to receive messages.

6 Conclusions and Future Work

We introduce a service discovery framework using activity policies. Existing research efforts on service discovery have focused on context-awareness and semantics support. However, due to the dynamic changes of semantics according to different contexts, the instances acquired in real environments might not be permitted or not exercise great influence on target users in certain situations. In order to resolve these problems, activity policies provide flexible and reusable resource constraints according to dynamically changing contexts. Thus, our approach enables developers to transparently make use of appropriate services despite heterogeneous environments.

In the future, we must refine activity policies into a dynamic range of environments as well as a home environment. Human activities will be divided to domain-independent and domain-specific activities. The predefined policies can be reused or customized by users, developers, or policy designers. Therefore, we need to provide them with some manners to simply manage the policies.

7 Acknowledgements

This research was supported in part by the Ubiquitous Autonomic Computing and Network Project, 21st Century Frontier R&D Program of the Ministry of Information and Communication (MIC) and by the Digital Media Lab. Support Program of the MIC and the Institute of Information Technology Assessment (IITA), in Korea.

References

1. A. Ranganathan, S. Chetan, R. Campbell, “Mobile Polymorphic Applications in Ubiquitous Computing Environments,” In Mobiquitous 2004 : The First Annual International Conference on Mobile and Ubiquitous Systems:Networking and Services, August 22-25, 2004 - Boston, Massachusetts, USA.
2. J. Sousa and D. Garlan, “Aura: an architectural framework for user mobility in ubiquitous computing environments,” In 3rd Working IEEE/IFIP Conference on Software Architecture, pages 29--43, 2002.
3. T. Broens, S. Pokraev, M. van Sinderen, J. Koolwaaij, P. D. Costa, “Context-Aware, Ontology-Based Service Discovery,” EUSAI 2004: 72-83.
4. C. Lee and A. Helal, "Context Attributes: An Approach to Enable Context-awareness for Service Discovery," In proceedings of third IEEE/IPSJ Symposium on Applications and the Intrnet, Orlando, Florida, January 2003.

5. Guanling Chen and David Kotz, “Context-sensitive resource discovery,” In Proceedings of the First IEEE International Conference on Pervasive Computing and Communications (PerCom 2003), pages 243-252, Fort Worth, TX, March 2003.
6. A. Harter, A. Hopper, P. Steggles, A. Ward, and P. Webster, “The anatomy of a context-aware application,” In Mobicom 99, ACM Press, pages 59–68, 1999.
7. A. Toninelli, A. Corradi, R. Montanari, “Semantic Discovery for Context-Aware Service Provisioning in Mobile Environments,” 9th MCMP 2005, Ayia Napa, Cyprus
8. J. E. Bardram, “Activity-Based Service Discovery – An Approach for Service Composition, Orchestration and Context-Aware Service Discovery,” CfPC 2004-PB-67, 2004, http://www.daimi.au.dk/~bardram/pvc/papers/absd.pdf
9. Z. Song, Y. Labrou, and R. Masuoka, “Dynamic Service Discovery and Management in Task Computting,” presented at First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services(MobiQuitous’04), 2004.
10. A. Ranganathan, S. Chetan, J. Al-Muhtadi, R. H. Campbell, M. D. Mickunas, “Olympus: A High-Level Programming Model for Pervasive Computing Environments,” In IEEE International Conference on Pervasive Computing and Communications (PerCom 2005), Kauai Island, Hawaii, March 8-12, 2005.
11. Z. Wang and D. Garlan, “Task-Driven Computing,” Technical Report CMU-CS-00-154, School of Computer Science, Carnegie Mellon University, May 2000
12. R. Masuoka, B. Parsia, and Y. Labrou, "Task Computing—The Semantic Web Meets Pervasive Computing," Proc. 2nd Int'l Semantic Web Conf. 2003 (ISWC 03), Springer-Verlag, 2003
13. Roman M. and Campbell R. H., “A User-Centric, Resource-Aware, Context-Sensitive, Multi-Device, Application Framework for Ubiquitous Computing Environments,” University of Illinois at Urbana-Champaign UiUCDCS-R-2002-2284 UILU-ENG-2002-1728, 2002
14. A. Ranganathan, and R. H. Campbell, “An Infrastructure for Context-Awareness based on First Order Logic,” Journal of Personal and Ubiquitous Computing, Vol. 7, Issue 6, Dec. 2003, pp. 353-364.
15. Sivashanmugam, K., Sheth A., Miller J., Verma K., Aggarwal R., Rajasekaran P., “Metadata and Semantics for Web Services and Processes,” Book Chapter, Datenbanken und Informationssysteme, Festschrift zum 60. Geburtstag von Gunter Schlageter, Publication Hagen, October 2003-09-26.
16. “Semantic Web projects,” http://www.semanticweb.org/
17. “New Racer Query Lanauge,” http://www.cs.concordia.ca/~haarslev/racer/racer-queries.pdf
18. “OWL,” http://www.w3.org/2004/OWL/
19. “Protégé,” http://protege.stanford.edu/
20. “Racer,” http://www.racer-systems.com/
21. D. Lee, S. Han, I. Park, S. Kang, K. Lee, S. J. Hyun, Y. H. Lee, and G. Lee, “A Group-Aware Middleware for Ubiquitous Computing Environments,” ICAT 2004, http://as.icu.ac.kr/www/pdf_paper/icat04-final.pdf.
22. L. Capra, W. Emmerich, and C. Mascolo, “CARISMA: Context-Aware Reflective Middleware System for Mo-bile Applications,” IEEE Transactions on Software Engineering, 29(10), 2003
23. P. Bellavista, A. Corradi, R. Montanari, and C. Stefanelli, “Context-Aware Middleware for Resource Management in the Wireless Internet,” IEEE Transactions on Software Engineering, Vol. 29, No. 12, December 2003