XVMF: An Extensible and Versatile Matchmaking Framework for Supporting Dynamic Application Adaptation in Ubiquitous Computing Environments

Kyungmin Lee, Dongman Lee, Insuk Park, and Seunghyun Han
Information and Communications University
119, Munjiro, Yuseong-gu, Daejeon 305-732, Korea
{kmlee, dlee, ispark, dennis}@icu.ac.kr

Abstract

This paper presents XVMF, an extensible and versatile matchmaking framework, which supports various service substitution mechanisms in dynamic application adaptation for ubiquitous computing environments. XVMF enables various substitution mechanisms to be easily developed by providing common abstracts for implementing a service substitution mechanism. It also allows an application or a user to select the most appropriate substitution mechanism with respect to its requirements by providing various substitution policies. The current Java implementation of XVMF supports three existing substitution mechanisms.

1. Introduction

In ubiquitous computing environments, an application or a task should migrate from one environment to another and adapt itself to the available resources in the new environment so that its users can perform their tasks seamlessly [8, 9, 11, 12]. One of the key concerns in application migration and adaptation is the heterogeneity of environments since it is not possible to assume that a service required by an application is always available in all the environments. To address this issue, an application needs to find and use a substitute of the required service if it is not available.

Identifying substitutability between services is essential for supporting dynamic application adaptation in heterogeneous environments. In general, existing approaches to identifying substitutability between services can be classified into three cases: *task based substitution*, a service can be replaced with another if both of them provide the same or similar functionality with respect to a user task [8, 12]; *semantic web based substitution*, if their inputs, outputs, pre-conditions, and/or effects (IOPEs) are compatible concepts with respect to semantic relations defined in ontologies [6, 7]; *subtyping based substitution*, if their signatures (argument types) and/or specifications (pre-/post-conditions) are matched with respect to various subtyping rules [5, 13].

However, the existing approaches cannot satisfy a full spectrum of application or user requirements in adaptation since they are designed with a specific application or user requirement in mind. In task based substitution, a service can be replaced with another that behaves differently in terms of user interfaces or supported features. However, users can be satisfied only if they can cope with such differences (e.g., Vi and MS Word). On the other hand, subtyping based substitution requires strict closeness between two substitutable services and thus enables transparent application adaptation. But, it unnecessarily limits the possibility of adaptation even if technology savvy users can afford to resolve some mismatches between services. Thus, we argue that a ubiquitous computing system should support a full spectrum of service substitution mechanisms to satisfy various application and user requirements in adaptation. To our best knowledge, however, there's no or little effort to incorporate various substitution mechanisms into a single framework.

In this paper, we propose an extensible and versatile matchmaking framework (XVMF) that supports various service substitution mechanisms in dynamic application adaptation in ubiquitous computing environments. XVMF defines six types of base elements required to implement a service substitution mechanism. A new substitution mechanism can be easily implemented by extending those base elements. XVMF provides various substitution policies with applications or users such that they can select the most appropriate one with respect to their requirements in adaptation. We implemented three substitution mechanisms with XVMF.

The rest of this paper is organized as follows. Section 2 presents the existing substitution mechanisms for dynamic application adaptation. Section 3 and 4 describe the design and implementation of XVMF,

Proceedings of the Fourth Annual IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOMW'06)
0-7695-2520-2/06 $20.00 © 2006 IEEE

respectively. Finally, we conclude the paper in Section 5.

2. Related Work

A service can substitute another if it preserves the latter's properties [5], which vary depending on application or user requirements. To check whether a service is substitutable to another, we need to define what kinds of service properties are required for identifying substitutability; to provide a way to formally specify those properties; and to give an algorithm that compares those properties. In the rest of this section, we describe the existing approaches to identifying substitutability between services.

In *task based computing* [11, 12], a *service supplier* provides an *abstract service* that comprises a user task. Substitutability between suppliers can be determined either by simply comparing their names if they are named after abstract services that they can provide; or by exploiting knowledge about substitutability between suppliers if they are named independently. The latter approach is taken and elaborated by [8, 9]. An application component can be replaced with a semantically similar one. To measure the semantic similarity between components, it uses an ontological hierarchy of components. These approaches allow services that have different features can substitute each other with the assumption that end-users can cope with such differences. However, it may distract novice users since they cannot do anything with a new substituted service. In addition, it is not applicable if an application does not allow users to directly interact with services.

Semantic web service initiatives aim at enabling automated discovery, composition, execution, and monitoring of web services [7]. A service's various properties are formally specified with OWL-S [6] in terms of shared concepts in ontologies. Two services are substitutable if the concepts of their inputs, outputs, preconditions, and effects (IOPEs) are semantically compatible. However, the granularity of a web service (one service with one operation) is different from that of the service in ubiquitous computing (one service with a set of operations).

Constructing or adapting a software system by (re-)composing existing components requires a mechanism that determines whether a component is substitutable to another. Behavioral subtyping rules [5, 13] guarantee that the substituting service preserves the behaviors of the substituted one so that the former does not cause unexpected results. However, this approach is so strict that it is likely to fail to find a substitutable service even if a user or a program can handle some differences between services.

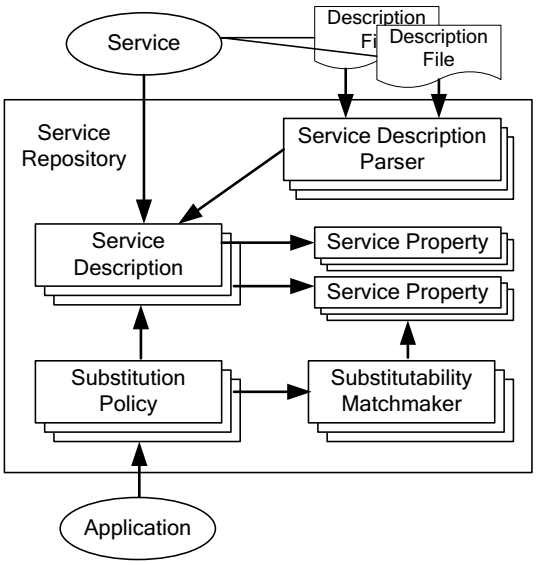

Figure 1. XVMF architecture

3. XVMF: An Extensible and Versatile Matchmaking Framework

3.1. Architecture

Figure 1 illustrates the base structural elements that comprise XVMF. It contains six types of elements: *Service Repository*, *Service Description*, *Service Property*, *Service Description Parser*, *Substitution Policy*, and *Substitutability Matchmaker*. We describe in detail the roles and interfaces of each element.

Service Property: It represents one of the properties that a service has. It consists of a name and value pair. Each substitution mechanism determines the properties required for checking substitutability between services.

Service Description: It represents the description of a service. It contains a list of service properties that a service has. It is uniquely identified with its name, which is usually the name of the corresponding service. It exports an interface for adding, removing, or retrieving a service's properties.

Service Description Parser: It is responsible for extracting service properties from service description or specification documents. For each description or specification format such as OWL-S [6] or JML [3], a cor-

Proceedings of the Fourth Annual IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOMW'06)
0-7695-2520-2/06 $20.00 © 2006 IEEE

responding parser should be provided. Each parser should implement the `parse()` method, which receives as an input a URI of a service's description or specification document and returns a list of extracted service properties.

Substitutability Matchmaker: It is responsible for comparing the descriptions of a required service and a substituting candidate and identifying the degree of substitutability between them. Each matchmaker should implement the `match()` method, which receives as an input the names of two service descriptions and returns the substitutability degree between them under its own matchmaking rules.

Substitution Policy: It is responsible for implementing a substitution policy that an application can choose. Each policy should implement `getSubstitutes()` and `update()` methods. The former receives as an input the name of a service description and returns the service's substitute(s) under its policy. The latter is called when a service description is inserted, removed, or modified. It finds and updates a substitutability relation between services in advance for fast retrieval of substitutes. An implementation of the `update()` method is optional since such pre-processing may not be required. The above two methods usually make use of one or more substitutability matchmakers to determine the degree of substitutability between services.

Service Repository: It maintains a list of service descriptions and substitution policies. It provides an interface for adding, removing, and retrieving them. It also exports an interface for services to (de-)register themselves as well as applications to retrieve a required service's substitutes under a specific substitution policy.

3.2. Developing a Substitution Mechanism

The base elements of XVMF provide common abstracts on which different substitution mechanisms can be easily implemented. Currently, we implemented with XVMF the three substitution mechanisms: task based, semantic web based, and subtyping based. The overall process of implementing a substitution mechanism with XVMF is described as follows (detailed implementation description of each substitution mechanism is presented in Section 4):

(1) Define service properties, i.e., their names and the data types of their values, required for checking substitutability between services. For instance, in semantic web based substitution, semantic concepts of input, output, precondition, and effect are defined as a service property, respectively; and their value type is a URI that refers to a concept in ontologies.

(2) Define and implement a service description parser, if required. As an example, we define and implement OWL-S parser that extracts semantic concepts of a service's IOPE from OWL-S.

(3) Define and implement a substitutability matchmaker. For the semantic web based substitution mechanism, OWL-S matchmaker is implemented to determine the substitutability degree between two services comparing their IOPE properties.

(4) Define and implement a substitution policy. For example, OWL-S policy is defined and implemented for supporting the semantic web based substitution policy.

3.3. Finding a Service's Substitute(s)

XVMF allows an application to select a substitution policy depending on its requirement in adaptation. A code fragment below shows how an application can use XVMF. Suppose that an application attempts to get a service instance in a new environment using a discovery service, but fails. Thus the application requests a service repository to find the required service's substitute(s) with indicating a specific substitution policy.

```
......
ServiceRepository sr = (ServiceRepository)
          Naming.lookup("ServiceRepository");
URI svc = "http://www.example.com/svcname";
URI plc = "http://www.example.com/plcname";
List l = sr.getSubstitutes(svc, plc);
......
```

A series of substitution policies can also be combined to increase the chance to find substitutes. The below code fragment shows how an application requests a service repository to find a required service's substitutes using multiple substitution policies.

```
......
ServiceRepository sr = (ServiceRepository)
          Naming.lookup("ServiceRepository");
URI svc = "http://www.example.com/svcname";
URI plc1 = "http://www.example.com/plcname1";
URI plc2 = "http://www.example.com/plcname2";
List plcList = new Vector();
policyList.add(plc1);
policyList.add(plc2);
List l = sr.getSubstitutes(svc, plcList);
......
```


Proceedings of the Fourth Annual IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOMW'06)
0-7695-2520-2/06 $20.00 © 2006 IEEE

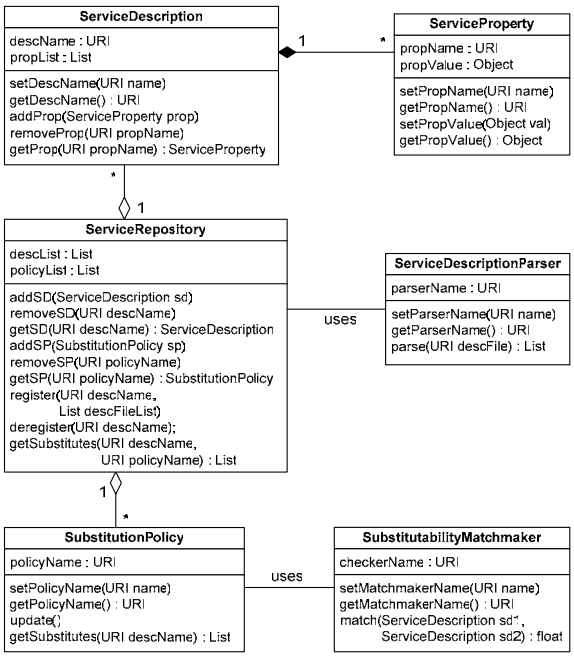

Figure 2. The XVMF Base Classes

4. Implementation

We implemented XVMF in Java and integrated it with Active Surroundings, a group aware middleware infrastructure for ubiquitous computing [4]. With XVMF, we implemented three substitution mechanisms mentioned earlier.

Each of the base elements of XVMF is implemented as a Java class, as shown in Figure 2. For unique identification of each instance, we use `URI` as their names. `Hashtable` is used for implementing a list in `ServiceDescription` and `ServiceRepository`. For each substitution mechanism, we implemented several classes that inherit from the base classes, as described below.

Task based substitution: To support task based substitution, we implemented three classes: `FunctionalSimilarityPolicy`, `FunctionalSimilarityMatchmaker`, and `Functionality`, by inheriting `SubstitutionPolicy`, `SubstitutabilityMatchmaker`, and `ServiceProperty`, respectively. In this case, we do not need to implement a parser because the service description is a simple pair of a service name and its functionality list. `FunctionalSimilarityMatchmaker` maintains a functionality ontology and adopts similarity measuring scheme from [8] to implement the `match()` method. `FunctionalSimilarityPolicy` maintains a `Hashtable` whose entry consists of a service and a list of its substitutes.

Semantic web based substitution: For the semantic web based substitution, we implemented several classes by extending base classes of our framework. `OWLSMatchmakingPolicy` class extends `SubstitutionPolicy` class. The `update()` method and `getSubstitutes()` of the class are implemented by the `SimilarityMatchingEngine` class of OWLS-MX [2]. The `match()` method of `OWLSMatchmaker` class which extends `SubstitutabilityMatchmaker` class is also implemented by using the `OWLSMXMatchmaker` class. The `OWLSParser` class of which the `parse()` method is provided by the OWL-S API [10] retrieves several service properties from OWL-S service descriptions. Those are `OWLSInput`, `OWLSOutput`, `OWLSPrecondition`, and `OWLSEffect`. `OWLSMatchmakingPolicy` provides *Exact*, *Plug-in*, *Subsumes*, *Subsumed-by*, and *Nearest-neighbor* matching capabilities based on these properties [2].

Subtyping based substitution: In the case of subtyping based substitution, we defined six `ServiceProperty` classes according to JML annotation. Those are `JMLInput`, `JMLOutput`, `JMLPrecondition`, `JMLPostcondition`, `JMLInvariant`, and `JMLConstraint`. The `parse()` method of `JMLParser` class and the `match()` method of `SubtypingMatchmaker` class are implemented using the ESC/Java2 library [1]. `SubtypingPolicy` has `Hashtable` which has pairs of a service and its substitute that meets the behavioral subtyping rules.

5. Conclusion

In this paper, we have proposed XVMF, an extensible and versatile matchmaking framework that flexibly accommodates various requirements of dynamic application adaptation. It allows various substitution mechanisms to be developed and applications or users to select their own substitution policies according to their requirements. We have implemented XVMF and three existing substitution mechanisms in Java.

XVMF allows a service's properties to be specified using various description languages since currently available service description languages have different aspects in mind. However, specifying a service with many different languages is burdensome. Moreover, attributes specified in these languages are not com-

Proceedings of the Fourth Annual IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOMW'06)
0-7695-2520-2/06 $20.00 © 2006 IEEE

pletely orthogonal. One possible solution is to devise a single description language that is easily extensible in terms of their attribute descriptions.

6. Acknowledgement

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

7. References

[1] D. R. Cok and J. R. Kiniry, "ESC/Java2: Uniting ESC/Java and JML: Progress and Issues in Building and Using ESC/Java2, Including a Case Study Involving the Use of the Tool to Verify Portions of an Internet Voting Tally System," in *Proc. of the International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS 2004)*, 2004.

[2] M. Klusch, B. Fries, M. Khalid, and K. Sycara, "OWLS-MX: Hybrid Semantic Web Service Retrieval," in *Proc. of the 1st International AAAI Fall Symposium on Agents and the Semantic Web*, 2005.

[3] G. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry, "JML Reference Manual (DRAFT)," Jul. 2004, (http://www.jmlspecs.org/).

[4] D. Lee, S. Han, I. Park, S. Kang, K. Lee, S. Hyun, Y. Lee, G. Lee, "A Group-Aware Middleware for Ubiquitous Computing Environments," in Proc. of the 14th International Conference on Artificial Reality and Telexistence (ICAT 2004), 2004.

[5] B. H. Liskov and J. M. Wing, "A Behavioral Notion of Subtyping," *ACM Transactions on Programming Languages and Systems, 16(6):1811-1841*, Nov. 1994.

[6] D. Martin, M. Paolucci, S. A. McIlraith, M. Burstein, D. McDermott, D. McGuinness, B. Parsia, T. Payne, M. Sabou, M. Solanki, N. Srinivasan, K. Sycara, "Bringing Semantics to Web Services: The OWL-S Approach," in *Proc. of the 1st International Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004)*, 2004.

[7] S. A. McIlraith, T. Cao Son, and H. Zeng, "Semantic Web Services," *IEEE Intelligent Systems*, 16(2):46-53, 2001.

[8] A. Ranganathan, S. Chetan, and R. Campbell, "Mobile Polymorphic Applications in Ubiquitous Computing Environments," in *Proc. of the 1st Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (Mobiquitous'04)*, 2004.

[9] M. Roman, H. Ho, and R. Campbell, "Application Mobility in Active Spaces," in *Proc. of the 1st International Conference on Mobile and Ubiquitous Multimedia (MUM)*, 2002.

[10] E. Sirin "The OWL-S API," (http://www.mindswa-p.org/2004/owl-s/api/).

[11] J. P. Sousa and D. Garlan, "Aura: An Architectural Framework for User Mobility in Ubiquitous Computing Environments," in *Proc. of the 3rd Working IEEE/IFIP Conference on Software Architecture*, 2002.

[12] J. P. Sousa, "Scaling Task Management in Space and Time: Reducing User Overhead in Ubiquitous-Computing Environments," Ph.D. Dissertation, Carnegie Mellon University, 2005.

[13] A. M. Zaremski, "Signature and Specification Matching," Ph.D. Dissertation, Carnegie Mellon University, 1996.

Proceedings of the Fourth Annual IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOMW'06)
0-7695-2520-2/06 $20.00 © 2006 IEEE