# A Comprehensive Middleware Architecture for Context-Aware Ubiquitous Computing Systems

Anjum Shehzad, Hung Quoc Ngo, S. Y. Lee, Young-Koo Lee*
*Real Time & Multimedia Lab, Kyung Hee University, Korea*
*{anjum, nqhung, sylee}@oslab.khu.ac.kr, yklee@khu.ac.kr**

## Abstract

*Ubiquitous computing is viewed as a computing paradigm where minimal user intervention is necessitated emphasizing detection of environmental conditions and user behaviors in order to maximize user experience. Context-awareness plays vital role in achieving such user-centered ubiquity. In this paper[1], we describe the desired characteristics of a middleware for context-aware ubiquitous computing. Four key issues are addressed: unified sensing framework, formal modeling and representation of the real world, pluggable reasoning engines for high-level contexts, and response to the real world. Our implementation experience indicates that a comprehensive approach throughout the system layers results in a flexible and reusable middleware framework.*

## 1. Introduction

Context-aware ubiquitous computing emphasizes on using context of users, devices, etc. to provide services that are appropriate to particular person, space and time. Every computing system dealing with the user should take into account human behavior in one way or the other to materialize ubiquitous computing experiences for him. As we all know that the role of middleware is to ease the task of designing, programming and managing distributed applications by providing a simple, consistent and integrated distributed programming environment; such middleware-based approach is quite appealing in context-aware ubiquitous computing [1].

A lot of work has been done in the area of context-aware computing, in which most of them are only concerned with one or more aspects in an ad hoc manner. Context Toolkit [2] uses the concept of widget to obtain raw contextual information from sensors and passes it either to interpreters or to servers for aggregation. Interpreters and servers use simple HTTP protocol for communication and the XML (name-value pairs only) as the language model for the context. In [3], graph based model for context aggregation and dissemination is proposed where contextual information sources are modeled as event publishers, while context-aware applications as event subscribers. Context Fabric [1] provides a distributed context-aware infrastructure to support the acquisition and retrieval of context data using an entity-relation style logical context data model, encoding data in XML and utilizing XPath as the query language. Gaia [4] is also a distributed middleware infrastructure supporting context-aware agents in smart spaces. It adopted a predicate model of context data encoded in DAML ontologies, and proposed that different logic reasoning and machine learning techniques can be adopted to support context inference.

Based on our knowledge, we have come up with a set of key issues; a context-aware ubiquitous computing system should tackle in order to successfully deploy in real life, namely unified sensing framework, formalized modeling and representation, supporting multiple reasoning approaches, and finally delivering the context to applications on semantic matchmaking basis. We have built and deployed a middleware infrastructure, CAMUS [5], addressing these challenges. We briefly describe those challenges in section 2 before explaining how our detailed architecture dealt with them in section 3. Finally, we present useful thoughts in section 4 and summarize in section 5.

## 2. Key issues in Ubiquitous Computing Middleware

Our middleware infrastructure addresses the above mentioned characteristics of context-aware computing systems and complements the existing middleware paradigms.

### 2.1. Sensing the Real World

Ubiquitous environments contain diverse range of sensors each utilizing its native access mechanisms and

[1] This research was supported by Ministry of Commerce, Industry and Energy, Korea.

output formats. This leads to potential problems and complexity in system design and implementation. Thus a mechanism is required, which serves to extract information from the heterogeneous sensors and present to the upper layers for deducing contexts, in a standardized and unified manner. Meanwhile, sensors are getting smaller and cheaper to be unobtrusively integrated into everything from shoes to coffee cups, and becoming autonomic sensing devices [6]. In such envisioned scenarios, the middleware system with a unified sensing framework will be able to synthesize data from all the sensors to form a more complete picture of the real world.

In our middleware architecture we introduce the concept Feature Extraction Agent (FXA), a software abstraction for sensing devices with two key characteristics:

- Hide the communication details and data polling frequency of sensors, and expose to upper layers with a Unified Access Pattern.
- Hide the specific algorithms for processing sensor data and the specific output format. Provide upper layers with the most descriptive features extracted from raw sensor data, in a common data structure of feature markup format [5].

## 2.2. Formal Modeling & Representation of Real World

The behavior of context-aware applications is mostly characterized by embedding the interpretation logic of context inside applications, which makes it difficult for other applications to reuse this information. Thus, in ubiquitous computing environments, applications need a shared understanding of context to communicate and transfer context effectively among them. Also, the applications demanding the contextual information from the environment may not have its prior knowledge, further emphasizing the need for common agreement of such information. All these problems of heterogeneity, independent interpretation, and need for interactivity leads us to think of a formal context model for efficient utilization of context in ubiquitous computing environment. The results are storing the context for later use and communicating it universally with other systems. Diverse entities like devices, users, and environment conditions are concepts in a certain domain and their inter-relation results in their association and dependency upon each other, e.g. user (John) is watching (inter-relation) television (device). Therefore, in this regard, we consider OWL [7] for formal modeling of context in our system because it allows us to define concepts and their inter-relationships e.g. describing person, devices, location etc., and to define instance data pertaining to some specific time and space, besides providing advantages of *expressiveness*, *knowledge sharing*, *logic inference*, *knowledge reuse*, and *extensibility* [7]. Also, OWL's meta-modeling language (RDF [8]) based approach makes possible for us to represent meta-information about sensors e.g. sensor access mechanism, quantized levels, feature list etc. Based on different entities e.g. PDAs, mobile phones, ambient displays, sound intensity, light, temperature, traffic, software agents, persons, groups etc, we categorize them, in our framework, mainly into agents, devices, environment, location and time [5], [9].

## 2.3. Reasoning for the Facts

Since not all information can be gathered from sensors, and sometimes the most interesting kinds of context are those that humans do not explicitly provide, demanding a need for context reasoning mechanisms. For instance, the current activity of a user could be inferred based on a combination of many other contexts, e.g. his location, his gestures, time of the day, environment status, etc [5]. Each reasoning mechanism has its own expressiveness, for example Description Logics (DL) is suitable for specifying terminological hierarchies while Spatiotemporal Logic is suitable concerning spatial-temporal sequence in which various events occur, and Bayesian Nets [10] are appropriate for learning the conditional probabilities of different events. Thus a middleware infrastructure needs to provide support for incorporating different reasoning mechanisms into the system, as well as specifying the appropriate mechanism for each context. This will facilitate not only the system internal modules to infer high-level context from low-level or predefined context, but also the applications to reason for their own application-specific context.

The following piece of code illustrates how to add and invoke a rule-based reasoner:

```
/* declare the prefixes for namespaces */
ContextReasonerManager.registerPrefix("conagnt",
rtmm.camus.vocabulary.contel.Agent.NS);
ContextReasonerManager.registerPrefix("env",
rtmm.camus.vocabulary.contel.Environment.NS);
ContextReasonerManager.registerPrefix("conloc",
rtmm.camus.vocabulary.contel.Location.NS);

/* add a new reasoner providing the rule file */
ContextReasonerManager.addReasoner("Location",
ReasonerType.GENERIC_REASONER, "etc/contel.rules");

/* declare some statements */
sms = new ContextStatement[] {PastLocationDescription,
hasLocation};

/* invoke the reasoner to do reasoning, providing the reasoner
name, the context data name and the required statements */
cdm.invokeReasoning("Location", "Data", sms);
```

In our middleware system, all reasoners are invoked through a unified interface. Such APIs make it possible to add and handle different reasoners as pluggable modules but, in return, it requires huge effort to come up with a uniform structure for different reasoning mechanisms. Currently, we are dealing with 5 different reasoning mechanisms mostly used in ubiquitous computing Description Logics, Neural Nets, Bayesian Nets, HTN Planning, and Fuzzy Logics.)

To provide more help to developers so that they can concentrate on developing rules or networks for reasoning and not be burdened with the low-level reasoning engine details, our middleware infrastructure defines wrappers for each Reasoner type. For example, a wrapper of Jena generic rule Reasoner allows the developer to easily add a new Reasoner just by declaring the *rule file name* and some *namespace abbreviations*.

### 2.4. Response to Real World Applications

Once the system senses real world correctly, saves the context in formalized manner and reasons intelligently, it must provide useful response to the real world to maximize the user computing experiences. Therefore, there is a need for an efficient delivery mechanism to filter out unrelated information and communicate the relevant contextual information to its respective clients.

The main motivation behind context delivery services in CAMUS is two fold:

- Provide a discovery and registration mechanism which can utilize the underlying contextual information's syntax as well as semantics in order to make more intelligent and accurate context service selection
- Incorporate dynamic and autonomous access-control mechanisms in the context delivery process to ensure privacy and overall integrity of the system

Keeping in view the requirements of the context delivery service and the representation scheme of the underlying data model for context, semantic web concepts for matchmaking [11] along with support for dynamic composition of context-aware services is being developed.

## 3. CAMUS Middleware Infrastructure

Our middleware architecture (figure 1) provides support for gathering context information from sensors in a unified manner, incorporating different reasoning mechanisms for deducing high-level context, and delivering appropriate contexts to applications as well as notifying the applications of context changes [5]. Here we mention our middleware architecture components, in sync with the key issues discussed in section 2.

### 3.1. Feature Extraction Agents

*Feature Extraction Agents* (FX Agents), or wrappers of sensors, extract the most descriptive features for deducing contexts in upper layers, sometimes attached with their semantic meanings and uncertainties. Then, *Feature - Context mapping layer* will perform the mapping required to convert a given feature into elementary context using some rules or reasoning mechanisms. In order to provide a generalized solution, our middleware infrastructure lets developer define and incorporate his meta-information for mapping, saved in the ontology repository. The meta-data relates to devices (D), sensors (S, including access mechanisms, feature list, etc.,) Feature - Context Labeling or Mapping (L), as well as the meta-information about the input, output and capabilities of pluggable reasoning modules (R).

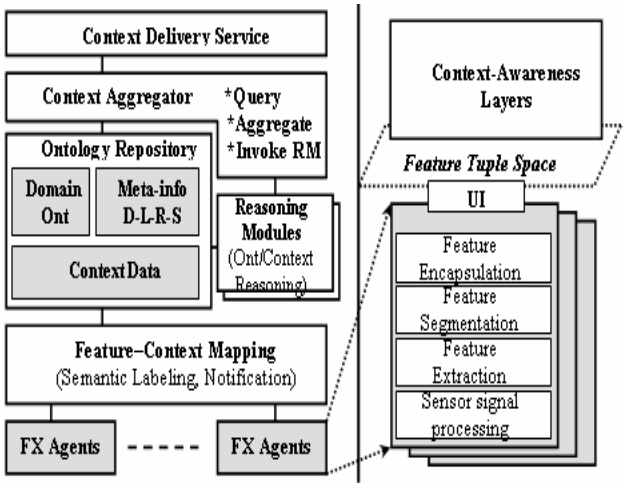


**Figure 1.** CAMUS Core Architecture

With the abstraction for sensing agents, the middleware infrastructure lets developer deploy any type of sensors in three steps:

1. Provide the native driver for communicating with the sensor hardware
2. Provide specific algorithms for extracting the most descriptive features from raw sensor data. Sensor Fusion can also be implemented at this step. These two first steps will modify the Feature Extraction Agent template provided by the Middleware System to have a new FX Agent for that sensor type.
3. Provide the meta-data describing new Feature Extraction Agent. This solution enables a dynamic mechanism for mapping between the real world information and the virtual world of context representation. For example, when a new RFID tag is attached to an object, the developer just needs to append new information describing this tag (e.g. the name of the object carrying the tag) to the meta-data

file of the RFID Readers, and no modification of the sensing modules is needed.

### 3.2. Context Repository & Query

The Context Repository provides the basic storage services in a scalable and reliable fashion and contains the *Domain Ontology* and *Context Information* along with *Meta-Information*.

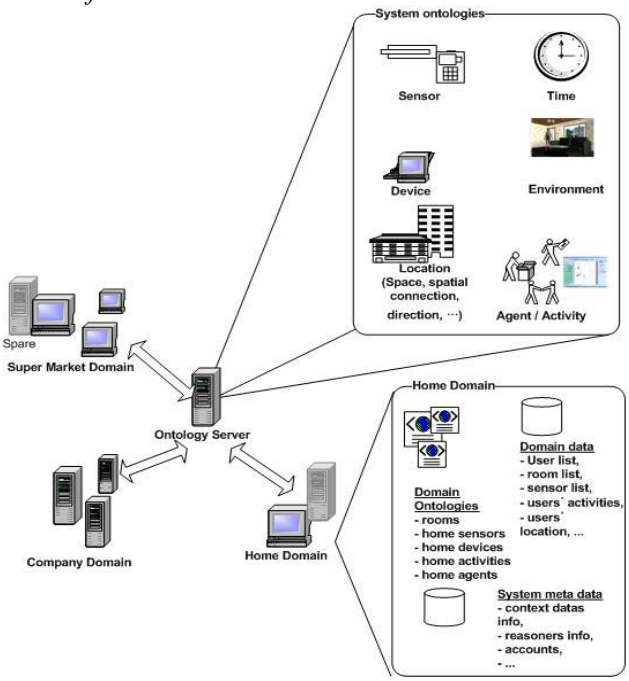


**Figure 2.** Context Repository Structure

*Domain Specific Data Model*: A ubiquitous computing system may consist of many subsystems running on various domains such as home domain, office domain, university domain, etc. Furthermore, many ubiquitous systems can collaborate with each other to build a large pervasive environment. The use of ontology can help sharing the knowledge about data among different domains and systems. However, such a distributed and dynamic environment requires an efficient mechanism to store and retrieve context data over multi-domain repository. As CAMUS uses OWL format to *store* context data, it maintains a meta-graph to manage the meta-data about all the domain repositories. Using OWL format, the Context Repository can be backed by some kinds of DBMS such as MySQL, or just use text files if the system needs to run on some resource-constrained environment. When handling OWL data using Jena library [12], each database can be considered as a group of models, where each model is a collection of contexts. The ontologies defining context data schemas have hierarchical structure, so each context data model itself is a sub-graph of the large graph combining all the ontologies. Consequently, it is feasible to build a meta-graph of all graphs in a ubiquitous environment. That meta-graph stores the information about the models of each domain, names and namespaces of the models, and especially the contexts provided by each model in a hierarchical structure. Context data can be retrieved by RDQL [13] queries. The queries are parsed into list of condition triples. Then the contexts mentioned in condition triples are used to search all the models which can provide those contexts from the meta-graph. After that, for each concerned model, all the related statements are extracted using the template statement built from the condition triples. Jena library allows us to integrate many statement sets into one model before executing the query. Because each Context Repository Manager module runs as a service, it can advertise itself as well as discover other Repository Manager services. Whenever it discovers a new repository, it will integrate the meta-graph of the new repository in its own meta-graph.

### 3.3. Structure of Reasoning Module

Reasoning Module in CAMUS includes multiple reasoners which handle the facts present in the repository as well as to produce composite contexts. The reasoners can provide the entailed knowledge not formally present in the repository using various kinds of logics to support inference. Moreover, since every context in CAMUS has probability property, many kinds of reasoning over uncertainty such as Bayesian inference or fuzzy logic can also be applied. Sample reasoning scenarios using different types of the above mentioned mechanisms are described in [5] and [9]. Here we focus on describing the structure and working of CAMUS Reasoning Module.

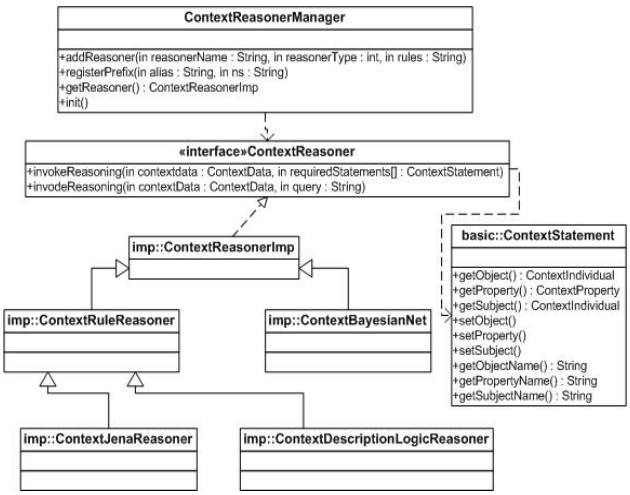


**Figure 3.** Class Diagram of Reasoning Engine

As depicted in figure 3, the *ContextReasonerManager* manages all the reasoners in

the system through a unified interface, Context Reasoner. All the reasoners will implement this interface. The most common method of a Reasoner is invokeReasoning, which does reasoning over a provided ContextData, to infer some statements, or to infer the new contexts required by an RDQL query.

Any service like Context Aggregator, or Client Mapping Service, can call the *ContextReasonerManager* service to add a new Reasoner, get an existing Reasoner, and then do reasoning by calling the *invokeReasoning*() of the reasoner. Developers just have to compose the rule sets, and decide the context data which should be used, and the middleware will take care of all other work from creating the reasoner to inserting the new inferred data into the repository.

### 3.4. Context Delivery & Aggregator Services

In our middleware framework, each context aggregator (analogous to web service) specifies the context it provides, by utilizing the concepts defined in the ontology repository. This standard schema sharing allows different kinds of entities to be described and utilized by delivery service to find useful services needed by the applications, thus, allowing a flexible mechanism for exchanging descriptive information of various entities.

It is clear that the capabilities of the context representation scheme can not be exploited to enforce access control over the information contents. The context delivery mechanism fills this void by incorporating dynamic policies at the services level as well as at the system level on the whole.

The foremost concern is to define access control policies that can be suitable in a pervasive environment. The major concerns are: the policies need to be dynamic in nature; the granularity of control to information needs to be identified; how to cope with changing policies based on the context at run-time; how can the clients trust the system while providing personal information to it for access and validation. Some of these issues can be overcome if autonomic access control techniques are employed in context delivery.

The services (context aggregators) utilize the registration interface to make their information known to the applications. Lookup interface enables the applications to find appropriate matching context providers. Policies/rules database contains the system level policies as well as optional aggregator services level policies and rules defining the requirements or conditions to access some specific service provided by the context aware middleware. This process is handled by the access control module.

The matchmaking module matches the appropriate service with the client provided the access control policies are not violated. Further breakdown of the context delivery module is represented in the figure 4. A further detail of this module has been cut out because of space limitation.

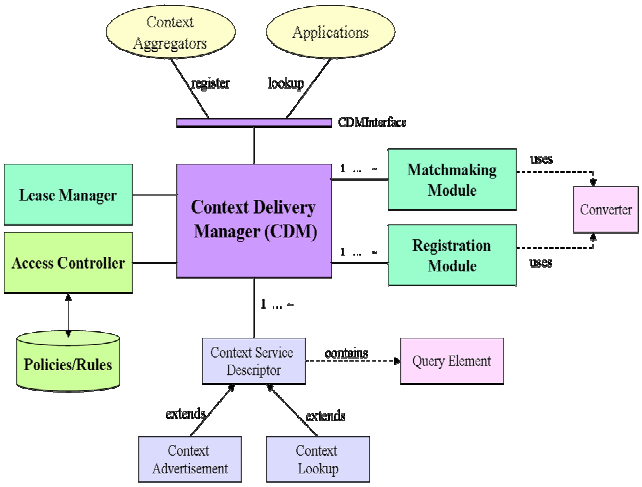


**Figure 4.** Context Delivery Service – Modular Breakdown

### 3.5. Runtime Composition

Composition of services is basically used in the workflow management systems such as [14]. The idea is very powerful and applicable to context-aware ubiquitous computing services when the context requested by user is not provided directly by single service but can be composed by combining several services in a flow. Since semantic matchmaking is being employed at the delivery service, we register service along with quantitative and qualitative semantics of its interface. Quantitative semantics is related to context service specification i.e. service name, operations/methods provided by the service through its exposed interface along with the inputs, outputs and exceptions of those methods, while qualitative semantics are dealt with excellence of the service i.e. its execution time, context freshness, reliability and availability semantics. An optional hardware used attribute can be used to show which hardware was used to gather the elementary context, e.g. location can be got by using with RFID, iButton, or even simple WLAN. Once context service is fully described (both quantitative and qualitative semantics), it can be then registered with the broker service. E.g. if the client is interested in user location and it is willing to provide user URI and expecting Location in terms of GPS location, then it can be defined roughly as:

Domain (CT) = Location Provider Service
Ontological class of required operation = UserLocation
Ontological class of required input = URI
Ontological class of required output = GPSLocation
Context Freshness < 5 Sec
Hardware Used = RFID || iButton

In this way, the client specifies its requirements in more expressive way and there are more chances to find suitable service as compared to simple search based on keywords.

## 4. Discussion

The benefits of using unified sensing framework approach are two fold. Firstly sensor access is unified through hardware abstraction layer and standardization which results in easy access for upper layers and masks sensor heterogeneity. Secondly, the use of features allows better description of different environment parameters than raw sensor values and features can be organized, stored and delivered in an efficient manner. For permanent storage of context data, OWL data is converted into relational DBMS by using the Jena framework API. This has certain performance limitations which made us believe the database storage schemes especially for OWL should be investigated along with different efficient query mechanisms to retrieve stored data. Similarly, providing different reasoning mechanisms to infer higher level demands a uniform data structure to incorporate information required by different reasoning engines. Applying some data mining and AI techniques into middleware needs to be considered, e.g. from the historical information of user location, user activity, the environment features, combine with user profile, some data mining algorithms can be used to mine the association rules which describe user preferences or user routine. Another example is that we can build the decision trees to predict future actions of user. Also, incorporating access control requires a lot of information inflow on behalf of the applications i.e., the applications are required to provide some credentials to match the policies. This process might be slow in case some mobile user just wants to retrieve general information e.g., weather, light conditions, humidity, goods available in the market etc from the context-aware system which are not subject to privacy constraints. In such scenarios, policies can be written to grant unhindered access to services that provide such contextual information. However, there is also a need to carefully define the data structure to represent policies and semantics so that the representation scheme facilitates these mechanisms. Concerns like dynamic policies, granularity of access control, coping with runtime police changes and providing certain level of trust of users can be dealt if autonomic access control techniques (out of scope of this paper) are employed in context delivery.

## 5. Summary

In this paper, important elements that comprise middleware for context-aware ubiquitous computing have been discussed. Context sensing, modeling and representation, context repository and query, pluggable reasoning modules, aggregators and delivery services, and runtime composition are all required components for a comprehensive context-aware middleware solution. The intermingling of all these components is necessitated to spotlight a comprehensive solution. Following a systematic approach makes CAMUS a flexible and reusable middleware framework.

## References


[1] Hong, J. I., et al., “An Infrastructure Approach to Context - Aware Computing”, HCI Journal, 2001, Vol. 16.

[2] Context Toolkit project, http://www.cs.berkeley.edu/~dey/-context.html

[3] Guanling Chen and David Kotz., “Solar: An Open Platform for Context –Aware Mobile Applications”, Proceedings of the First International Conference on Pervasive Computing (Pervasive 2002), Switzerland, June, 2002.

[4] Anand Ranganathan, Roy H. Campbell, “A Middleware for Context -Aware Agents in Ubiquitous Computing Environments”, ACM/IFIP/USENIX International Middleware Conference, Brazil, June, 2003.

[5] Hung, N.Q., Shehzad, A., Kiani, S. L., Riaz, M., Lee, S., “A Unified Middleware Framework for Context Aware Ubiquitous Computing”, EUC2004, Japan, Aug. 2004.

[6] Managing Care Through the Air, http://www.spectrum.ieee.org/WEBONLY/publicfeature/dec04/1204net.html

[7] W3C Web Ontology Working Group, “The Web Ontology language: OWL”, http://www.w3.org/2001/sw/WebOnt/

[8] Klyne, G., Caroll, J. J., “Resource Description Framework Abstract Concept and Syntax”, W3C Recommendation, 10 Feb. 2004.

[9] Anjum Shehzad, N. Q. Hung, Kim Anh Pham, Sungyoung Lee, “Formal Modeling in Context Aware Systems”, Workshop on Modeling and Retrieval of Context, CEUR, ISSN 613-0073, Vol-114, 2004.

[10] Korpipaa, P., Koskinen, M., Peltola, J., Makela, S. M., Seppanen, T., “Bayesian approach to sensor-based context awareness”, Personal and Ubiquitous Computing, Vol. 7, Issue 2, July 2003, pp. 113-124.

[11] Trastour, D., Bartolini, C., Gonzalez-Castillo, J., “A Semantic Web Approach to Service Description for Matchmaking of Services”, HP Labs Bristol, HPL-001-183, 2001.

[12] “Jena: A Semantic Web Framework for Java”, http://jena.sourceforge.net/

[13] Andy Seaborne, “RDQL – A Query language for RDF”, http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/

[14] F. Casati, S. Ilnicki, L. J. Jin, V. Krishnamoorthy and M. C. Shan, “eFlow: a Platform for Developing and Managing Composite e-Services”, HP Laboratories Palo Alto, HPL-2000-36, March 2000.