SEMANTIC INFRASTRUCTURE FOR A UBIQUITOUS COMPUTING ENVIRONMENT

BY

ROBERT EDWARD MCGRATH

B.S., University of Illinois at Urbana-Champaign, 1976
M.A., University of New Hampshire, 1980
M.C.S., University of Illinois atUrbana-Champaign, 1985

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduage College of the
Univesity of Illinois at Urbana-Champaign, 2005

Urbana, Illinois

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

UMI Number: 3199083

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI®

UMI Microform 3199083

Copyright 2006 by ProQuest Information and Learning Company.

All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CERTIFICATE OF COMMITTEE APPROVAL

University of Illinois at Urbana-Champaign
Graduate College

July 22, 2005

We hereby recommend that the thesis by:

ROBERT EDWARD MCGRATH

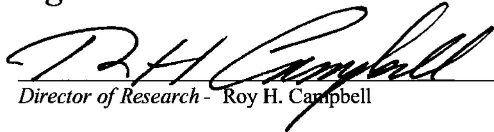
Entitled:

SEMANTIC INFRASTRUCTURE FOR A UBIQUITOUS COMPUTING ENVIRONMENT

Be accepted in partial fulfillment of the requirements for the degree of:

Doctor of Philosophy

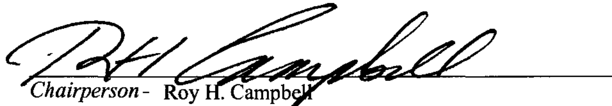
Signatures:



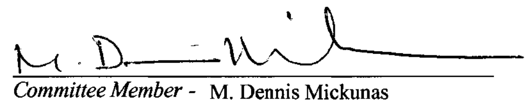
Director of Research - Roy H. Campbell

Head of Department - Marc Snir

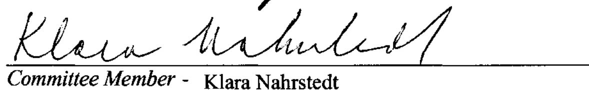
Committee on Final Examination*



Chairperson - Roy H. Campbell



Committee Member - M. Dennis Mickunas



Committee Member - Klara Nahrstedt

Committee Member - Robin Kravets

Committee Member -

Committee Member -

* Required for doctoral degree but not for master's degree

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Abstract

This thesis investigates one of the fundamental problems for Ubiquitous Computing: managing metadata to enable resource discovery. This work presents a flexible and general model of metadata, and proposes to use *ontologies* as a formal language for the metadata of a Ubiquitous Computing Environment. Ontologies provide a common language for metadata among diverse and autonomous entities and spaces in the Ubiquitous Computing Environment.

Description Logic is introduced as a formal language for metadata. Description Logic can be used to represent the concepts of a model, and the formal semantics can be used to maintain logical consistency.

This thesis presents important locality principles: a Ubiquitous Computing Environment is a local space, which needs a dynamic "working set" from the hypothetical universe of possible devices, services, and entities. This concept is familiar from other contexts (e.g., memory management), but has not been recognized or used in ontology based systems. These principles provide a key insight that led to development key algorithms for managing ontologies in a Ubiquitous Computing Environment: an algorithm for composing two ontologies, and semantic queries on ontologies.

The composition algorithm is the key to dynamically updating a local ontology to maintain a working set of concepts from a larger universe of ontologies. The composition algorithm exploits the formal semantics of the ontologies to maintain logical consistency when combining ontologies from several sources.

The logical relations defined in the ontology enable *semantic queries* which can discover not just exact matches but logically related concepts. An improved algorithm for semantic matching is proposed, which extends and refines previous work from the literature, using the formal semantics of the ontology to define what concepts are similar to each other.

A prototype Ontology Service was built using standards and software from the "Semantic Web". The prototype implements algorithms for managing ontologies. The prototype was evaluated. In other work, the prototype Ontology Service was ported to Gaia.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

This project would not have been possible without the support of my family, friends, teachers, and colleagues.

Thanks to all my teachers. Thanks to Professor Bruce Schatz and Professor Duncan Lawrie. Without your encouragement, I would never have begun. I would like to pay tribute to the memory of Professor Lance Cannon and Professor Dan Slotnick. You taught me better than you knew.

I owe a great debt to my friends and colleagues of the Scientific Data Technologies group at the National Center for Supercomputing Applications. Thanks for letting me dedicate so much time to this work. Special thanks to my good friends and mentors Mike Folk, Nancy Yeager, and Elena Pourmal. You guys taught me many important things that aren't in the formal curriculum.

Finally, and most important of all: this work would not have been possible without the woman I love, Patricia S. Taylor.

Thanks to all of you for sticking with me.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Acknowledgments

This research is supported in part by the National Science Foundation grant NSF 98-70736, NSF 9970139, and NSF infrastructure grant NSF EIA 99-72884. Important aspects of this study used software from Iona Technologies Inc. and University of Manchester [9].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table of Contents

Chapter 1. Introduction 1

1. Introduction 1
2. Example Scenario 3
3. Sketch of a Solution 6
4. Summary and Plan of Thesis 9

Chapter 2. Hypothesis 13

1. Introduction 13
2. Some Definitions and Assumptions 13
   - 2.1. The Target: Ubiquitous Computing in a "Smart Space" 14
   - 2.2. What is Smart (or Stupid) About Spaces? 15
   - 2.3. What Spaces will be built? 17
3. Assumptions about the Ubiquitous Computing Environment 18
   - 3.1. The Distributed Object Environment 18
   - 3.2. The Role of Standards in a Ubiquitous Computing Environment 19
4. Problem Statement: Resource Discovery in a Ubiquitous Computing Environment 20
   - 4.1. Diversity: the Need for a General Model for Metadata 20
   - 4.2. Why Discovery is Challenging in a Ubiquitous Computing Environment 22
   - 4.3. Analysis of the example 23
5. A General Model for Metadata 25
   - 5.1. The Meta-model 26
   - 5.2. Important Requirements for the Metadata Language 27
6. Ontologies: The Silver Bullet? 28
   - 6.1. Definition of "Ontology" 29
   - 6.2. Ontologies in Ubiquitous Computing Environment 30
   - 6.3. Example Use of an Ontology 32
7. Related Work 34
   - 7.1. Standard Interfaces and Services 34
   - 7.2. Discovery, Matchmaking, and Brokering 35

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

7.3. Ontologies in Ubiquitous Computing Environments 38
7.4. Requirements for Ubiquitous Computing Environments 40
8. Summary 43
Chapter 3. The Model and Approach 44
1. Introduction 44
2. Overview: Ontologies for Ubiquitous Computing Environments 45
2.1. Overview of the Use of Ontologies 45
2.2. A Hybrid Model for Service Registration and Discovery 47
3. Methodology for Developing Ontologies: Divide and Conquer 51
3.1. Locality: a Fundamental Design Principle for Ontologies 53
3.2. A Hierarchy of Domain Ontologies 53
3.3. Upper Ontologies 54
3.4. Locality of Use: A Local Working Set 55
4. A Fundamental Design Pattern: A Specialization of the "Proxy" Pattern 56
5. Encoding Ontologies: A Formal Language for Metadata 58
5.1. Languages for Metadata 60
5.2. Description Logic 63
6. How Description Logic is Used to Represent and Manage Ontologies 71
6.1. Representing Ontologies in Description Logic 72
6.2. Using Description Logic to Implement Proofs About Ontologies 74
6.3. Validation of an Ontology 75
6.4. Summary 77
7. Summary 77
Chapter 4. Algorithms for Managing Ontologies 79
1. Introduction 79
2. An Algorithm for Loading and Composing Ontologies 80
2.1. Introduction and Justification 80
2.2. Constraints on Ontologies 81
2.3. "Hints" for Composition 82
2.4. The Composition Algorithm 83
2.5. Some Properties of the "Compose" Operation 85

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.6. Summary of Composition 88
3. Design and Algorithms for Semantic Query 89
3.1. Definitions and Background 89
3.3. Improved Definition of Semantic Match 97
4. Summary 112
Chapter 5. Implementation: Development and Use of Ontologies 114
1. Introduction 114
2. A Model for Real World Objects in a Ubiquitous Computing Environment 114
2.1. The Basic Model: People, Places, and Things (PPT) 115
2.2. The Model 117
2.3. Grounding in An Upper Ontology 121
2.4. Use of the Abstract Classes of the PPT Model 126
3. Metadata Encoding Using the Semantic Web Standards 131
3.1. The Semantic Web Stack 132
3.2. Encoding the Ontology in XML 136
3.3. Heuristics for Knowledge Capture 138
3.4. Summary Deployment and Usage 139
4. Examples of DAML+OIL Ontologies 140
4.1. The Ontology for Person, Place, and Thing (PPT) 141
4.2. Examples of Domain Ontologies 146
4.3. Other Ontologies 163
5. Summary 166
Chapter 6. Implementation: A Prototype Ontology Service 168
1. Overview of the Prototype 168
2. Overview of the Ontology Server 170
2.1. The Current Ontology and Knowledge Base 170
2.2. The Ontology Service Interface 171
3. OntoKB: a Generic Interface to a Knowledge Base 174
3.1. Implementation of Automated Reasoning 176
3.2. Implementation of the OntoKB 185
4. Implementation of the Composition Algorithm 190

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.1. Input to the Composition 191
4.2. Composing the Ontologies.. 191
4.3. Axioms (Optional Assertions) 194
4.4. Return the Results.. 194
4.5. Example: Composing the Thing Ontology 194
5. Implementation of Semantic Queries 199
5.1. The Query Interface.. 200
5.2. Description of the Query (Input).. 200
5.3. Query Execution 203
5.4. Result Sets and Information Returned to the Caller 206
5.5. Discussion.. 206
6. Summary.. 207
Chapter 7. Evaluation of the Prototype 208
1. Introduction.. 208
2. General Design and Usability: Integration into Gaia 208
3. Evaluation Criteria: Correctness and Performance.. 209
3.1. Evaluating Ontologies.. 209
3.2. Evaluating Implementation of the Algorithms 210
3.3. The Test Environment and Methodology.. 211
3.4. Evaluating Queries and Query Answering 211
4. Results 1: Correctness.. 214
4.1. Correctness of the Composition Algorithm.. 214
4.2. Correctness of Queries.. 214
5. Results 2: Performance of the Composition Algorithm 244
5.1. Basic Overhead: Load the Empty Ontology.. 244
5.2. Load and Compose the Person, Places, and Things Ontology 245
5.3. Scale-up: A Larger Ontology.. 246
5.4. Summary.. 249
6. Results 3: Performance of the "Semantic Query" Algorithm.. 250
6.1. Overhead.. 250
6.2. The Service Description Ontology.. 250

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

6.3. The Composed People, Places, and Things (PPT) Ontology 254
6.4. Summary 257
7. Discussion 257
7.1. Correctness: Precision and Recall of the Queries 258
7.2. Performance of the Service 259
Ch. 8. Summary and Comparison to Other Work 260
1. Overview 260
2. Ontologies in Information Systems and Knowledge Engineering 262
3. Intelligent Agents and Brokers 264
4. The Semantic Web and Semantic Web Services 267
5. Ontologies in Ubiquitous and Pervasive Computing 268
6. Prototypes for an Ontology Service 269
7. Summary 270
Ch. 9. Conclusion 271
1. Summary of Thesis 271
2. Implications 275
2.1. Addressing the Semantic Interoperability 275
2.2. Standards for Semantic Infrastructure 278
2.3. Integration of Models and Languages 280
3. Conclusion 281
References 282
Appendix 1: Listings 303
Listing 1 303
Listing 2 309
Listing 3 310
Listing 4 311
Vita 314

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 1. Introduction

1. Introduction

Computing devices are becoming smaller, cheaper, and ubiquitous. Soon there will be orders of magnitude more computers than people on the planet. This Malthusian fact demands fundamental changes in the way humans and computers interact. One of the revolutions will be the emergence of Pervasive and Ubiquitous Computing Environments; environments that intelligently monitor and manage the virtual and physical objects of the space, including the people, in order to create new and better human experiences [141, 257].

This thesis investigates one of the fundamental problems for Ubiquitous Computing: managing metadata to enable resource discovery. This work builds on earlier work in distributed systems, databases, and artificial intelligence. I present a flexible and general model of metadata, and propose to use *ontologies* as a formal language for the metadata of a Ubiquitous Computing Environment. Ontologies provide a common language for metadata among diverse and autonomous entities and spaces in the Ubiquitous Computing Environment.

It is widely recognized that what makes "smart" spaces smart is the existence of a rich "context" and the ability to interact with "real" objects (as opposed to digital objects) (e.g., the Context Toolkit [47], the iRoom [65, 260], the CoBrA system [27, 31, 32], and many others [38, 70, 123, 138, 188, 207, 212, 229, 230, 238, 257]). In this environment, the real world objects of interest include people (users and others), things (such as furniture, merchandise, and medical materials), and the space itself. These physical entities are increasingly available to the digital world via improved sensors (including cheap video) and electronic tags. The interaction requires sensing and tracking physical objects, and also will require adequate models of the objects of interest. Since interactions with physical objects occur in a physical space, a spatial model of places is needed as well.

It seems clear that true Ubiquitous Computing will require infrastructure support, such as Gaia [206] and other similar systems [27, 30, 47, 65, 123, 133, 138, 229, 230]. Distributed object technology such as JINI [53] and CORBA [104] provide platforms for implementing such systems. This technology provides the foundation for *how* to implement the system, it is now necessary to investigate *what* must be implemented, and how to manage the complexity of the system.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A Ubiquitous Computing Environment must manage interactions among a diverse and heterogeneous set of entities, which include people, places, and things, as well as software components, services, and devices. There are multiple activities, which may require different models of the real world. Different users, applications, and contexts have different models of what is important about the same entities; models may refer to disjoint or overlapping sets of entities, and may define different attributes for the same entities. Furthermore, the environment is open and constantly changing, and must deal with specialization and localization. This diversity is not the result of bad design; in fact, it is the essence of good design.

In order to deal with this diversity, the Ubiquitous Computing Environment needs abstract logical models to manage the entities. Components should be designed to manipulate abstract entities, which are bound to real objects as the software is deployed and activities occur. The abstract models provide a level of indirection that enables applications, components, and services to adapt and evolve as the environment changes, and to operate in different local spaces.

Given the heterogeneous and dynamic environment, the configuration of a Ubiquitous Computing Environment is complex and volatile. A given environment can be a constantly changing set of devices and software; from many sources; with multiple uses of the same components. A central challenge for a Ubiquitous Computing Environment is to manage this diversity with minimal human intervention. The environment needs to be able to automatically reason about the current configuration, application context, and system policies, e.g., to compose services while maintaining consistency, security, and Quality of Service constraints. These algorithms will need to operate on metadata about entities from many sources.

Metadata is used to exchange information between services, applications, and users, in the form of messages or data structures. Metadata defines a language for statements about objects, e.g., statements that describe entities and relationships in the computing environment. However, given the diversity of the entities and abstractions, there will inevitably be many languages and dialects of metadata.

In recent years, a variety of experimental frameworks and systems have laid the foundation for Ubiquitous Computing Environments (e.g., [1, 70, 119, 205, 256]). An evaluation of these efforts indicates that there is a significant "Tower of Babel" problem: the metadata of the system is diverse and difficult to maintain e.g., [144]). This problem is shared by any open,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

decentralized system, including the World Wide Web. Several lines of research have converged to propose using *ontologies* as a lingua franca for heterogeneous information systems.

The ontologies of the "Semantic Web" implement this concept with XML languages suitable for distributed systems such as the World Wide Web or Grid [14, 126, 159]. While not specifically designed for a Ubiquitous Computing Environment, the Semantic Web technology can be applied in context-aware, Ubiquitous, and Ubiquitous Computing.

Unlike the World Wide Web, a Ubiquitous Computing Environment is a local environment: the ontology for a local time and place is a subset of a (hypothetical) universal ontology. As the local system evolves, the set of entities and activities change, so the set of concepts and relations must evolve as well. Therefore, an ontology for a local system must be updated automatically as entities enter and leave the local area. Essentially, the system should maintain a local "working set" of concepts from the universe of all ontologies. This will require algorithms for automatically using and managing ontologies in a dynamic real-time local environment.

In a Ubiquitous Computing Environment, ontologies will enable new and improved services; with ontologies integrated into protocols to define the content of messages, i.e., the "vocabulary" of a local system or context. In this approach, statements in the metadata refer to one or more ontologies that define the terminology used. The ontologies enable producers and consumers to agree on terminology and to translate when equivalences are known. To make this work, there must be a standard infrastructure to manage ontologies.

This thesis develops some of the foundations for implementing this approach. A general and flexible model of metadata is developed, which is implemented using the *ontologies* of the "Semantic Web" [14, 159]. This chapter introduces the problems with an example, followed by a summary of the challenges and the plan of the thesis.

2. Example Scenario

The basic scenario is a person entering and leaving local spaces in a Ubiquitous Computing Environment, carrying handheld computer devices. As users move throughout the environment, they want to accomplish tasks using the handheld devices and the devices and services of the local space. A handheld device implements services for the user, collaborating with the local space and other network services. The room has sensors, I/O devices, services, and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

physical objects of interest (including people), and the handheld device itself has capabilities, such as buttons, a display, speaker, microphone, and software components.

To make this problem more concrete, consider the case of a smart appliance, a coffee mug which has a computer, wireless networking, and software to communicate with services in the local environment. The role of the mug is to act as a user agent: to represent the user's preferences to the environment, and to translate information into actions to satisfy the user's cravings. As the user moves through cities and buildings, the mug should collaborate with local services to order coffee through the net, give directions to a service point, and/or provide other services.

The key to this service is for the coffee mug to query the local environment to discover local instances of services that provide coffee. There are many possible services, including coffee pots, break rooms, vending areas, restaurants, tea carts, delivery services, and so on. There is also a large and baroque universe of products, including many variants of coffee drinks, as well as alternatives such as cocoa, and tea; and there are a number of generic categories of service that might be useful for satisfying the goal, such as "hot drink", "beverage", or even "food". In addition, there are many condiments for coffee drinks, including creamers, sweeteners, and other possibilities.

A given environment has only a tiny subset of this universe of possible services and products, and that set may evolve rapidly. Therefore, it is not likely that it will be possible to compile a single, universal database for all possible services and locations. Similarly, there may be a large population of potential users, with a diverse and changing array of preferences and goals. So it is not possible (nor desirable) for a service to have a comprehensive database of its users.

In this scenario, a person carries a mug around, entering various smart spaces. When the user requests coffee (perhaps with a button or a gesture), the mug will connect to local infrastructure, and make a query such as, "find a cup of coffee, or else another hot beverage". Ideally, the infrastructure will give a list of services near by, along with enough information to enable the mug to fill the request (e.g., by directing the user to a service point nearby.).

However, what is more likely to happen will be disappointing. Sometimes, the local environment will not have anything registered that matches the precise query, and so will return "no match found". For instance, even if a vending machine was near that advertised "coffee with

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

cream and sugar", "decaf", and "tea", these might not be recognized as a match for either "cup of coffee" or "hot beverage". Another possible result would be for the environment to reply with information such as "Vending machine A serves 'small coffee', 'medium coffee', and 'large coffee.". Unfortunately, the mug might not recognize these as instances of "cup of coffee", and may not recognize that "Vending Machine" is a kind of service provider.

These misunderstandings are examples of a "Tower of Babel" problem: wherever there are heterogeneous, autonomous entities that need to communicate, this challenge must be addressed [144, 147, 262]. This problem is encountered by many components of a Ubiquitous Computing Systems, including mobile devices, local services, and middleware components; and is the same for systems designed with centralized portals, distributed agents, or peer-to-peer architectures.

Similar challenges are faced in the other direction as well, when services in the local space seek to interact with the resources of the handheld device, e.g., to send data to be displayed by the handheld device. There are a great variety of mobile devices, with many different collections of capabilities. Not only does a local space need to collaborate with many possible devices, it should work with new models of devices, never before seen in the space. Again, discovery and binding operations must be done in this case: the space must discover the capabilities of the device, and bind generic services to the specific capabilities of the available devices, translating as needed.

A fundamental challenge to this process is the heterogeneity of the Ubiquitous Computing Environment. Each space has a different collection of services and devices, and the configurations may change. In addition to different attributes, the entities may have alternative but equivalent "vocabularies" for their interfaces and protocols.

Resource discovery is often implemented manually, e.g., by presenting a human user with a list of services to pick from. Similarly, devices and services might be configured into a space by manually written scripts (e.g., [23]). These manual methods are hardly "seamless". They are difficult to keep up to date, and do not scale to very large numbers of entities. It is important for this discovery process must be automated as much as possible [144].

In a Ubiquitous Computing Environment, the problem is complicated by the need for the models and protocols to include physical objects in the environment. When a handheld device enters the room, sensors and access control devices will detect both the device and the person

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

carrying it. A proxy for the person must register with the space, along with a description of him with respect to the local environment. Similarly, other objects in the space and the space itself need to register and be described.

3. Sketch of a Solution

"Any problem in computer science can be solved by an extra level of indirection."[1]

The solution to the problem is for software to implement its operations using an abstract model of an entity that delivers the required service. When the user request is executed, the abstract entity must be bound to a real device in the current environment. This binding may require a mapping or translation to generate the correct request to the selected entity.

The physical world contains objects, events, relations, and other concepts of interest. Software components of a Pervasive Computing Environment are designed to manipulate abstract entities, e.g. through service requests to abstract interfaces which are bound to real objects as the software is deployed and activities occur. The abstractions reflect (explicit or implicit) models of the entities of the environment. The abstract models provide a level of indirection that enables applications, components, and services to adapt and evolve as the environment changes, and to operate in different local spaces.

The dynamic binding can be accomplished by using a flexible and extensible metadata language for describing entities, including software, devices, events, and physical objects. The metadata is an intermediate representation used to exchange information between services, applications, and users. The metadata provides the needed level of indirection between diverse components of the environment.

Figure 1 presents a three-tiered scheme, in which the computational model is implemented by mapping the abstract concepts of the model to statements in the metadata. The statements are, in turn, mapped to entities or concepts in the physical environments. Thus, there are two levels of mapping that need to be defined.

---

[1] The original source of this quote is not precisely known. Lampson et al. [135] attribute this phrase to David Wheeler, citing the authority of Roger Needham.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In this environment, even basic operations such as service registration and look up must manage, combine, and use metadata from many sources. In short, the Ubiquitous Computing Environment faces similar challenges of fusing multiple databases or knowledge bases.

Information systems, such as large multiple database systems ([259]) or the World Wide Web ([147, 210]) face a serious challenge: independent information sources use incompatible vocabularies and conceptual structures. Fusing data from multiple sources requires mapping from one set of key words to another, and from one database schema to another, which is laborious and sometimes impossible (e.g., [17, 186, 193]). Even when possible, pair wise translation between N sources requires N^2 translators, which cannot scale up. This challenge has been called the "Tower of Babel" problem.

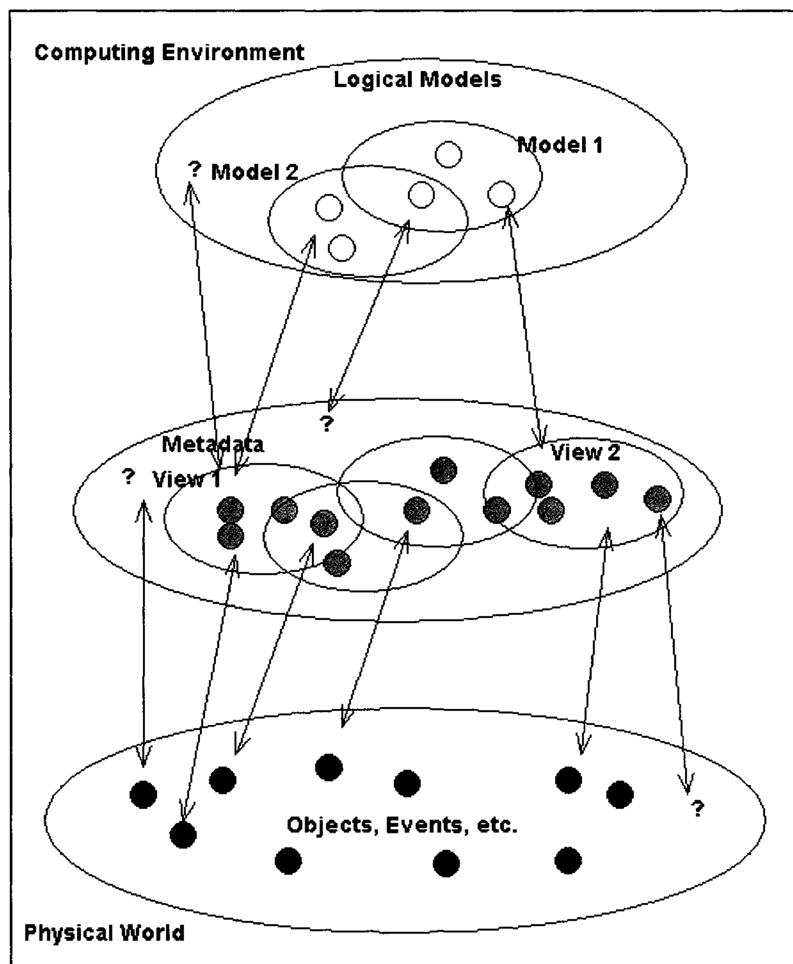


Figure 1. A Meta-Model of Ubiquitous Computing

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Artificial Intelligence research has faced a similar problem among multiple Knowledge Representation systems. Different expert systems use different conceptual organizations and vocabularies, which makes it difficult to integrate knowledge from multiple Knowledge Bases [88]. This problem has been especially critical for Autonomous Agent systems, in which software agents must conduct sophisticated conversations with independently developed agents—"strangers"—that may well have different vocabulary and conceptual schemes [87, 129, 226].

This problem has been addressed by attempting to develop a standard and flexible model for metadata, expressed in a formal language, to provide a lingua franca for heterogeneous information systems. These have been termed *ontologies*.

The Semantic Web defines an XML language (the Web Ontology Language (OWL) [249][2]), which has a formal semantics. Essentially, the XML maps to statements in formal logic, which can be used to prove correctness and other properties for the XML. This language is used to encode *ontologies*, which are vocabularies that define classes, attributes, and relations. In this thesis, ontologies are used to implement the metadata for the Ubiquitous Computing Environment.

In the example above, an ontology for the local space would define the classes of entities in the local space in one or more XML documents. For example, the ontology would define the set of services, service items, and attributes.

The ontologies provide a common language that enables the handheld and the entities in the space to implement the discovery and translation needed to bind services. The ontology can be used to implement services analogous to those of a class browser in a reflective software system because the information in the ontology is similar to the class hierarchy of an object-oriented programming language, except it is not limited to software objects.

The ontologies can help solve these interoperation problems by implementing a *semantic match*, which defines a set of classes that are *similar* to the target, and therefore potentially

---

[2] This study used the DARPA Agent Markup Language, Ontology Interchange Language (DAML+OIL) [42]. The DAML+OIL language has now been standardized as the Web Ontology Language (OWL) [248]. OWL is a superset of DAML+OIL, with similar theoretical properties.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

substitutable. The ontology defines which classes are synonyms, either explicitly declared or deduced from the Knowledge Base, and also classes that are "partly equivalent": more specific or more general interfaces than the target, and classes that are logically compatible with the target, according to the Knowledge Base.

The Semantic Web languages and Knowledge Bases are a necessary foundation, but they are not sufficient to manage the metadata for a Ubiquitous Computing Environment. There is a need for standard services to manage *ontologies* (rather than Knowledge Bases). This thesis presents an initial prototype for these services, built on top of the Semantic Web standards.

In order to deploy ontologies in real systems, a broad array of challenges must be met. A complete implementation must include:

1. conceptual modeling to define metadata
2. a language or languages for metadata
3. encoding(s) of the languages
4. algorithms for manipulating the languages (e.g., correctness, equivalence, answering queries)
5. population and maintenance of running systems

Several groups are investigating application of technology from the "Semantic Web" ([14, 251]) to context-aware computing and Pervasive Computing (e.g., the CoBrA project [27, 31, 32] and others [5, 38, 150, 229, 230]). This thesis is provides a comprehensive study of this approach, including a prototype implementation and several original algorithms. This approach is the foundation for a general and flexible infrastructure to manage metadata in the environment.

The Ontology Service developed in this thesis is a prototype of the type of infrastructure that will become a standard part of future Ubiquitous Computing Environments. In the future, this work will converge with related work from several areas to define common standards in Web Services [2, 19, 44, 57, 58, 143, 154, 155, 159, 184, 189, 218], the Semantic Grid [34, 35, 43, 75, 139, 213, 224], and Pervasive Computing Environments [27, 29, 30, 38, 229, 230].

4. Summary and Plan of Thesis

This thesis develops some of the foundations of a general and flexible infrastructure for managing metadata for a Ubiquitous Computing Environment.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 2 presents some preliminary definitions and assumptions. First, critical aspects of the Ubiquitous Computing Environment are defined. In particular, these environments are distinguished by the need to manage physical objects, i.e., objects that have not direct connection to the digital environment, such as people and furniture. The models of the environment must include physical objects as well as software and hardware.

Chapter 2 considers questions such as "what kinds of spaces should be built" and "what objects are interesting?" The answer is that practically everything might be of interest in some part of the environment, so the model must be general but specialized to different spaces.

This leads to the central research topic: how to manage metadata for these diverse and dynamic systems. A general model for metadata is presented, and *ontologies* are introduced as a formal language for metadata.

Chapter 3 develops theoretical foundations for implementing these models. The abstract model and the physical world are linked via one or more mappings. This mapping is an essential level of indirection. The linking is implemented by defining formal languages and standards for *metadata. Description Logic* is introduced as a formal language for metadata. Description Logic can be used to represent the concepts of a model, and the formal semantics can be used to maintain logical consistency.

Chapter 3 present important locality principles: a Ubiquitous Computing Environment is a local space, which needs a dynamic "working set" from the hypothetical universe of possible devices, services, and entities. This concept is familiar from other contexts (e.g., memory management), but has not been recognized or used in ontology based systems. These principles provide a key insight that led to development of the prototype and algorithms in the later chapters.

Chapter 4 builds on the foundations to develop algorithms for managing ontologies in a Ubiquitous Computing Environment: an algorithm for composing two ontologies, and semantic queries on ontologies.

The composition algorithm is the key to dynamically updating a local ontology to maintain a working set of concepts from a larger universe of ontologies. The composition algorithm exploits the formal semantics of the ontologies to maintain logical consistency when combining ontologies from several sources.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The logical relations defined in the ontology enable *semantic queries* which can discover not just exact matches but logically related concepts. In Chapter 4, an improved algorithm for semantic matching is proposed. This algorithm extends and refines previous work from the literature, using the formal semantics of the ontology to define what concepts are similar to each other. This kind of query is critical for heterogeneous services, agents, and other entities to successfully interoperate in a Ubiquitous Computing Environment.

The creation and management of ontologies is a difficult problem. Chapter 4 presents a method for developing ontologies for a Ubiquitous Computing Environment. Ontologies are developed piecewise, reflecting the natural conceptual locality of the conceptual model. These pieces can be dynamically composed to create customized metadata for spaces, applications, and contexts.

To illustrate this process, a basic conceptual model is developed for the physical objects of the system. At the highest level, the model considers People, Places, and Things, i.e., the answer to the questions "who?", "where?", and "what?" This model will be extended and specialized for different application environments. Three examples are developed to illustrate the used of the conceptual model: Hospital, Shopping, and Library.

In Chapter 5, concepts from the example models are encoded into ontologies, using the DAML+OIL XML language, from the so-called "Semantic Web". This language has a formal semantics, which makes it possible to prove that the ontologies are logically consistent. These ontologies can be used as input to implement the algorithms developed in Chapter 4.

Chapter 6 presents a prototype Ontology Service, which manages ontologies and operations on ontologies. The Ontology Service builds on standards and software developed for the "Semantic Web", to implement the algorithms developed in Chapter 4, using ontologies encoded in DAML+OIL XML.

Chapter 7 presents an evaluation of the prototype, to show that it is correct and measure the performance. In other work, the prototype Ontology Service was ported to Gaia, and has enabled experiments with enhanced services [152, 200, 203].

Chapter 8 reviews related work and the contributions of the thesis. A Ubiquitous Computing Environment is a local, real-time system that must operate seamlessly and without human intervention. This thesis extends and adapts previous work, building on foundations from

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

information systems, broker- and agent-based systems, and the World Wide Web, to solve the problems of Ubiquitous Computing.

Finally, Chapter 9 closes with a summary and brief discussion of future work.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 2. Hypothesis

1. Introduction

Chapter 1 presented the challenges presented by the diversity of the entities in a Ubiquitous Computing Environment. This thesis investigates the key problem of metadata for this environment. Specifically, I claim that ontologies should be used as a common language for metadata for Ubiquitous Computing Environments. Ontologies will be incorporated into the standard services and protocols of the infrastructure to enable improved resource discovery, interoperability, and other advance services.

This chapter presents basic definitions and assumptions about the Ubiquitous Computing Environment. Section 2 presents a general definition of a Ubiquitous Computing Environment. Section 3 presents assumptions about the software environment and an abstract pattern for representing physical objects is presented.

Section 4 presents the problem of resource discovery in a Ubiquitous Computing Environment, and introduces the need for "semantic discovery". The diversity of the environment is analyzed, and it is argued that the Ubiquitous Computing Environment requires many models and a flexible and general language for metadata.

Section 5 defines the general model for metadata. Section 6 introduces *ontologies*, which are used to implement a formal language for metadata and gives an example.

Section 7 summarizes previous work related to this thesis. Additional related work is discussed in detail in the following chapters.

2. Some Definitions and Assumptions

To begin, it is necessary to consider what a Ubiquitous Computing Environment is, and how it is distinct from other systems. This discussion will begin with abstract description of the envisioned environment. The focus will be narrowed to a subset of environments, "smart spaces", specifically, localized areas (such as a room, or building) that are specifically designed for an activity.

This thesis targets a limited but rich set of environments and technologies. First, it is assumed that there is an open, platform independent distributed object infrastructure, such as

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Gaia [205]. Second, it is assumed that the spaces of interest are "purpose-built", i.e., they are dedicated to a primary activity. Examples of such spaces include meeting rooms, libraries, hospitals, and retail stores. Because the environment is dedicated to specific application, it is assumed that the objects of interest are known and can be modeled. These assumptions are discussed in this section.

2.1. The Target: Ubiquitous Computing in a "Smart Space"

A Ubiquitous Computing Environment is a complex linking of real world and digital objects. Sensors, actuators, and interactive interfaces create events and data that correlated to real world objects and events through computer models of the entities and environment [141, 257]. A "ubiquitous" environment has many possible realizations. This section considers a subset of Ubiquitous Computing Environments, "Smart Spaces". A smart space is a localized area, augmented by an infrastructure and services.

A "smart" space is a *physical* space, which is *augmented* by virtual space(s). The physical space contains objects of interest, including people, documents, and furniture. The physical space may also be characterized by conditions of interest, such as light level. The virtual space contains many kinds of virtual objects, including mobile and static devices, conventional software objects, but also representatives of the physical objects of interest. The smart space seeks to intelligently monitor and manage the virtual and physical objects of the space, in order to create new and better human experiences.

There are a very large number of spaces that might be made into smart spaces. These range from single rooms, to buildings, campuses, and "smart cities". These spaces differ in scale (both in area and number of users), and most important, have many different uses and users. The latter point is important because a smart space is "smart" only in the eyes of its users and with respect to the tasks they are trying to accomplish. So, the diversity of spaces implies that there will be a diversity of behaviors required to make them smart.

It would be very difficult to analyze all possible spaces in a single study. Furthermore, many spaces are themselves complex, multi-purpose environments. For example, an airport supports a complex array of activities. A "smart" airport would have to be smart in a variety or ways, about a variety of tasks, and would have to deal with the interactions of the different activities. Note, too, that an airport might be better modeled as a set of smaller, more specialized spaces, such as, ticket counter, baggage claim, waiting area (in which users might work, sleep,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

eat, etc.) and so on. The complexity of the airport stems not from its scale, but from the number of different activities that it supports.

This thesis will consider a tractable but still interesting subset of possible spaces: spaces that are designed and dedicated for a few built-in purposes. A classic example of such a space is a meeting room: it is dedicated to a few related tasks, and a smart meeting room need only be smart about meetings. Because they have limited functions, these spaces can be analyzed and modeled more easily, and a smart space can be developed more easily. There are many examples of such spaces that are interesting and of practical value. As noted above, large, complex spaces may often be decomposed into sets of such dedicated spaces.

2.2. What is Smart (or Stupid) About Spaces?

In order to understand this problem, it is necessary to understand what we want spaces to be smart about, and what they are currently stupid about today. One important reason why computers seem "stupid" is that they usually do not understand the users' intentions and other critical aspects of the context of the activity. For example, without knowledge of the user's location and head position, displays of visual information are often non-optimal, even with excellent software and hardware. The problem is not beyond solution, but the computer often does not have the context information it needs.

It is widely agreed that in any situation a key part of smartness is the availability of relevant context information (e.g., [47, 65, 212]). Even smart virtual services can be rendered extremely stupid when they lack information about the context of the activity. For example, a word processor is an extremely smart program within its virtual world. It manages files and file formats, with clever storage management and caching. It has extensive knowledge of text layout, fonts, and appearances. It has many clever features to support collaboration among multiple users, templates for controlling styles, and personalized preferences.

However, when faced with the "real world", the word processor is rendered stupid as a rock. It knows little about printers (usually just a channel) and nothing at all about the actual paper documents produced. The result can be a beautiful screen that produces illegible garbage on paper. Similarly, the word processor knows little about the human being that uses it. For instance, it has no knowledge of the location (or even presence) of the user, of the lighting conditions, or of the user's current focus of attention. The lack of this information may result in

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

sub-optimal displays, strange (to the human) dialogs, and complex menus of so-called "preferences" and "accessibility options" that must be manually adjusted.

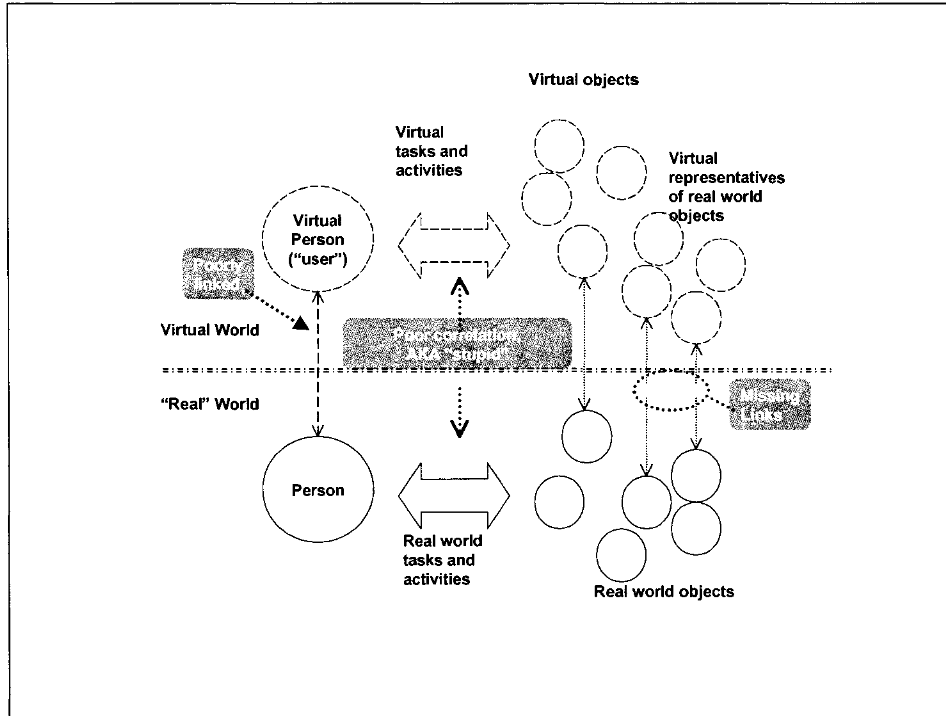


Figure 2. Overview of a "Smart" Space

Figure 2 illustrates this idea. In the lower half of the diagram is the "real world", in which people live and act, perceiving and interacting with objects and events in their environment. In the upper half of the diagram is a "virtual world", in which people are represented by objects, and interact with digital objects through user interfaces. The virtual world may contain representatives (proxy objects, or wrappers) of many real world objects. For computers or computerized objects, the virtual representative can interact with the physical object, learning its state and requesting actions. The system potentially can be intelligent about such objects. But other objects of interest, including people, are not usually wired in this way, so the virtual object and the physical object may be poorly correlated and poorly coordinated. The virtual world has no *linkage* to these relevant real world objects and events. The result is a virtual world that is comparatively stupid, because its activities can take no account of real world conditions.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A central tenet of context-aware computing is that a smart space (potentially) can be intelligent about any objects that it can identify and track, because these objects can be incorporated into the model of the context of the activity. In the case of smart spaces, this context involves not only the state of virtual objects, but also the recognition of any and all physical objects (events, etc.) that are the relevant for the task(s), establishing the appropriate virtual views of this world, and tracking the status and behavior objects and the user's activities. To obtain this context, the system(s) of the smart space must be able to incorporate objects of the physical world into the virtual "world" of the smart space or application.

From the point of view of constructing a smart space, we must not only recognize objects and their positions, but we must recognize their relevance, and specifically, their relevance to the task at hand, e.g., their *affordances* [169]. Within the general model described above, this may be done by linking specific physical objects to appropriate virtual representatives. This linkage includes not only the existence of virtual proxies, but also:

- Detection and tracking of objects
- Ability to *access* objects appropriately, e.g., *read* a book or article
- Ability to manipulate objects appropriately, e.g., *open* a door

It should be clear that different applications have many different requirements, so the *relevant* objects and *appropriate* methods depend on the purposes of the user in the space, i.e., there may be many views of the same physical and virtual objects.

A specific smart space would be smart because it is *preconditioned* to be ready to support certain kinds of tasks. This space would be designed to present the kinds of affordances needed by the activities it seeks to support (e.g., meetings, health care, or archival research).

2.3. What Spaces will be built?

For purposes of this study, the focus will be on applications that have the following characteristics:

- The space is a complex space in which people interact with both physical objects and information systems (i.e., virtual objects).
- These interactions are critical to the activity.
- The space is designed to support the activity (as opposed, say, to a general-purpose public space that may sometimes be used for the activity).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- Large amounts of digital information about the physical objects already exist, e.g., in the form of an inventory database.

The first two criteria define applications that have an interesting combination of physical and virtual activity, and therefore are worth investing effort to make the space smarter. The third criterion defines a sort of "atomic" space, from which multi-purpose spaces might be constructed. The fourth criterion is a practical desideratum: an activity which is already significantly digitized eliminates some of the routine work of creating and populating databases. In addition, the existence of such databases usually means that there is a well-developed model of the application, which is crucial for developing smart spaces.

3. Assumptions about the Ubiquitous Computing Environment

In the previous section, a Ubiquitous Computing Environment was defined to be a physical space that is enhanced by a large number of computers, sensors, actuators, and other equipment. People move into, through, and out of the environment, carrying portable devices, data, and software. The Ubiquitous Computing Environment integrates these components to create an enhanced, "smart" space, in which applications can automatically adapt to the local environment and the local space can detect and adapt to activities.

The environment is heterogeneous, the devices, spaces, applications, and components are developed independently. Devices and software are developed in a particular environment, yet then should be able to work in other spaces. A space is designed for specific uses, but must be able to recognize and handle new devices and software without extensive manual intervention.

This environment requires a computing infrastructure that is general enough to be deployed everywhere, yet capable of adapting to many different local configurations with no human intervention. The infrastructure requires a set of standard services (interfaces) and protocols, an also tools for creating and managing spaces.

3.1. The Distributed Object Environment

This thesis builds on previous work using CORBA to construct secure component-based infrastructure [205]. The CORBA standard provides key building blocks, including:

- definition (IDL) and implementation of objects (GIOP), and related services
- Naming and Trading services
- Event service

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- CORBA security framework

Other distributed object systems, such as Java RMI and JINI [53], .NET [106] or OGSI [232] provide similar building blocks.

The 2K and Gaia projects have developed secure, reflective infrastructure based on CORBA [205]. These extensions are ideal for constructing ubiquitous computing for smart spaces. The Gaia infrastructure provides essential building blocks, especially reflection (for dynamic configuration and reconfiguration) and integrated Quality of Service management.

3.2. The Role of Standards in a Ubiquitous Computing Environment

Standards play an important role in the infrastructure for Ubiquitous Computing Environments, as in any open, dynamic system. Standards are required for entities to interoperate with each other in an open system, and they can lower the cost and risk of system development. Furthermore, a "ubiquitous" computing environment requires universal standards and protocols.

When possible, the Ubiquitous Computing Environment should be defined as a specific profile of one or more general standards. While much of the Ubiquitous Computing Environment can be built as combinations of "atomic" services provided by standards, it will be necessary to define additional standards for Ubiquitous Computing.

In recent years, other services have emerged that play a critical role in Ubiquitous Computing. These include software to manage sensors, Discovery Services, and other infrastructure and is needed to support security, management of local spaces and the construction of applications [23, 105, 205]. These services must be integrated with the core distributed object system.

A Ubiquitous Computing Environment is characterized by large numbers of sensors that provide information about the physical environment. These sensors detect and identify (physical) objects and conditions, allowing new kinds of applications to be constructed. Sensors must be incorporated into the infrastructure, through software toolkits such as the Context Toolkit [47], or iSpaces [65, 123, 190].

Ubiquitous Computing systems are characterized by a dynamic state, with devices and people coming and going, new objects appearing, and so on. In this environment, it is important to discover and maintain knowledge of the current state of the system (i.e., which objects and services are *currently available*). "Discovery protocols", such as JINI Discovery Service ([53]) provide this sort of information [144]. These services must be incorporated into the infrastructure

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

as well. This thesis develops a key part of the infrastructure, to improve advertising and discovery.

4. Problem Statement: Resource Discovery in a Ubiquitous Computing Environment

A Ubiquitous Computing Environment is an open system, in which the components are heterogeneous and autonomous. Entities must spontaneously collaborate (i.e., dynamically construct chains of services) to accomplish the required tasks. To accomplish this, consumers and producers must *discover* the current configuration of the system and capabilities of services and components.

This process has been termed *discovery* or *matchmaking* [144, 196, 222, 231]. *Matchmaking* seeks to find sets of candidates that might fulfill the required and desired service request. The match is based on the request, available entities and services, and other criteria, such as quality of service.

Discovery and matchmaking can involve several related activities: advertising, querying, and browsing. In each case, the parties exchange structured records describing the offered service (advertising, response to query) or the desired service (querying). The exchange may be manual (browsing), real-time (a query to discover the current local state of the system), persistent (a standing query, e.g., notification). The exchange may be a push (advertisement, notification), pull (query), or some combination. In all cases, it is critical that the data is filtered, to select a set that best matches the intentions of the parties. These use cases are summarized in Trastour, et al. [231] and earlier work [144].

4.1. Diversity: the Need for a General Model for Metadata

The Ubiquitous Computing Environment must manage a diverse and heterogeneous set of *entities*, which includes physical objects such as people, places, and things; as well as software components, services, and devices. This section summarizes the challenges presented by this diversity.

The Ubiquitous Computing Environment needs abstract logical models which can be realized as computational models to manage the environment, users, and activities. Clearly, even for the same environment there may be many different models, depending on the task, as well as different models for different environments. Even for seemingly trivial cases, different uses or users might well need alternative viewpoints, levels of granularity, and behavior.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The abstractions of the model must be represented in the system by machine-readable data, e.g., statements that describe entities and relationships in the computing environment. Because this data is secondary (in that it is *about* the objects of interest), it is usually termed *metadata*. The metadata is an intermediate representation used to exchange information between services, applications, and users. The metadata also provides the needed level of indirection between diverse components of the environment.

The Ubiquitous Computing Environment is a diverse set of environments, tasks, and users, which can be viewed in many different ways. This diversity is not the result of bad design, in fact, it is the essence of good design. The models must reflect this diversity, and also must deal with specialization and localization. The system is open and constantly changing, so the models must be able to adapt and evolve.

The models will need to meet a number of requirements that stem from the nature of Ubiquitous Computing Environment itself. First, the environment is a continuously operating, real-time system. It must run with minimal human intervention, and must automatically adjust to changes in users, activities, and configurations. This will require robust infrastructure, and as much autonomous configuration as possible.

Second, the environment is an open and dynamic system, its components are heterogeneous and come from autonomous sources. The Ubiquitous Computing Environment must be able to accept "new" devices and software, and autonomous environments must be able to interoperate when needed.

Third, the Ubiquitous Computing Environment is decentralized. While there may be some centralized or wide-area services, individual rooms will contain local services, and many devices will have their own local state. In any case, a complete model of the environment would be a model of almost everything, which is beyond reach. Therefore, it will be necessary to decompose the problem into smaller, tractable pieces, and to compose solutions from multiple smaller models.

These facts require the computational model to be adaptable and robust as the system changes. Even more important, the system needs to be general enough to unify diverse local environments, but must be able to be specialized and localized for particular needs. Furthermore, it is unlikely that all the decentralized, autonomous components of the system can tightly coordinate their models, even if they wanted to. The model must deal with the "Tower of Babel"

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

problem: entities from different sources and environments cannot be assumed to have a common design. The metadata for the Ubiquitous Computing Environment must be general and flexible, in order to express the concepts of diverse models in a mutually intelligible form.

4.2. Why Discovery is Challenging in a Ubiquitous Computing Environment

Standard registry services and discovery services can locate services, objects, and interfaces, and can deliver and launch implementations. But, in order to use these services, the consumer must understand the semantics of the objects and interfaces, i.e., what the object does and how to use it. This problem is fundamental to any open distributed system, including the World Wide Web [210], Agent-based systems [226], and digital libraries [147].

It is important to note why a conventional registry such as LDAP [255] or the CORBA Naming Service [175], and so-called "discovery" protocols such as Salutation [187] or JINI Discovery Service [220] are inadequate. These systems answer questions by matching records or strings.

While keyword or string matching is relatively easy and efficient to implement, and is widely used, e.g., in web search engines. String matching performs well in limited cases: essentially when the vocabulary is controlled. However, string matching is known to yield poor results for heterogeneous data and queries, unless the concepts rather than the words can be matched [33, 68, 136, 210]. Improved matching requires structuring the data (and queries), e.g., as networks of objects and relations.

Record-oriented query systems such as LDAP [255], Globus Resource Specification [73], or JINI [219] have similar limitations to simple string matching. For example, in LDAP, queries are treated as paths, which can always be reduced to long strings.

A second problem is that most registries have limited ability to dynamically add new types, i.e., to update the scheme. A key problem is that dynamic update requires propagating logical constraints, which is quite difficult unless the data is highly structured. In some systems, all applications must be recompiled when the data scheme changes—clearly not feasible in a Ubiquitous Computing System.

A third problem is that registries have limited ability to deal with incomplete or uncertain information. Conventional databases employ a "closed world" rule, positing that the absence of a record is equivalent to non-existence. In a system based on this logic, inconsistent or incomplete

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

information produces errors or wrong answers to queries. This is obviously a significant problem for a highly dynamic and distributed system.

One important implication of these shortcomings is that the entities in the Ubiquitous Computing Environment cannot necessarily know how to construct queries or how to interpret results. Successful discovery requires that the producer and consumer reach agreement via advisement and queries. Since the environment and available services is dynamic, a service cannot always know what attributes to advertise (what consumers may ask), and an application does not know what queries to send (what targets may exist).

Answering questions about a Ubiquitous Computing Environment will require more sophisticated queries, that seek to match "conceptually equivalent" items, which may have different values for equivalent concepts (i.e., different terminologies). This process has been termed "semantic matching" [144, 196, 222, 231].

In general, a semantic match is usually considered to be a *conceptual match*: objects that are "logically equivalent" according to some model. Clearly, the match depends on the information available and the intentions of the participants. In general, the goal of a "semantic query" is to provide answers to questions in the face of incomplete and uncertain information, and it must work for many different users, who may have different logical views of the same information.

4.3. Analysis of the example

Chapter 1 introduced an example problem, a smart mug that finds coffee in a Ubiquitous Computing Environment. This example illustrates the diversity of a Ubiquitous Computing Environment, even for a simple case. There are many ways that coffee might be delivered, and many variants of "coffee" that might be found, and the available options are different in different locations, and change over time. Also, the population of potential users is large and diverse, as well as mobile.

This task can be viewed from several points of view. First, the user (or user agent) has a goal to satisfy in different local environments. Second, services want to advertise in specific environments for an unknown population of potential consumers. Third, the system seeks to meet these goals through general mechanisms.

User's View. The goal of the user is to obtain a cup of coffee. In fact, a range of solutions *might* satisfy the goal.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- A very exact match, e.g., fresh coffee with the exact condiments specified
- Something close enough, e.g., decaf, or substitute condiments
- Sometimes something similar might be accepted, such as tea or even cold water.
- There might be something new, e.g., a new product or service provider

User Agent View. The coffee mug acts as an agent for the user. The agent should be able to issue a generic query that *works everywhere* and *continues to work as the environment evolves*. In this context, "work" means that it finds local services that are "close enough" (to satisfy the user). The mug should be able to work in most environments without special action or customization. In all cases, the result should be better than the user could achieve unaided, and should also be capable of finding "interesting" or "surprising" results: non-obvious or totally new items.

The agent (or other application) must interrogate the current local Ubiquitous Computing Environment to discover services and devices that can fulfill this request. Note that the user and the application may have never been in the current environment before, and in any case, the environment constantly evolves. The user and the application cannot know in advance what services and devices are present, so they must rely on the infrastructure to provide candidate services that can meet the request. This can be viewed as a simple case of dynamically composing a service.

Service View. A service, such as a vending machine, wants to advertise to diverse customers. The services in a local environment want to advertise their *specific* services in a *generic* way. This advertisement has work for a variety of user agents.

When a device or service is introduced into the system (e.g., when it is started or enters), it will automatically register with the infrastructure. The registration will advertise the network address, software interfaces, physical location, and other important facts about the new service. In some cases, a service might be capable of providing several classes of service, which it may or may not explicitly advertise in its registration.

The devices and services originate from multiple vendors and providers, and cannot know what the environment will contain, what users or applications may be present, or that the requests will be. Services must rely on the infrastructure to route service requests, based on registration and advertisements. From the perspective of the services, they must be able to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

advertise their service(s) in a way that can be used in many environments and by many applications.

System View. The system infrastructure needs a general and flexible model of services, along with a binding to real services. This requires generic protocols with many specific localized uses. Middleware and brokers need to translate among heterogeneous agents, services, etc., in different environments.

The system is continuously changing, so the advertisement, query, and response protocols must be robust in the face of new configurations, new versions and models of devices and services, and from local space to local space. For this reason, a query should be a *description of the required service*, rather than a request for a specific entity, or for a particular model or version. The application should not attempt to "open a connection to Printer-7", because this will be very fragile. Rather, the application should describe the desired service ("print here"), and the infrastructure should provide candidate services that can meet the request in the current environment.

The matchmaking process calls upon the *infrastructure* to bridge the differences between the perspectives of the user and multiple services, to compose a set of services that meet the goals of the user; i.e., the infrastructure needs to implement protocols for *semantic discovery* or *matchmaking*. In these protocols, services and devices advertise a description of the service(s) they provide, and users, applications, and agents request services in the form of a description of the required service. The infrastructure seeks to match these descriptions, to provide one or more configurations that can meet the request.

The common denominator between these views is the system metadata. The following section develops a general model for metadata and metadata languages.

5. A General Model for Metadata

"Any problem in computer science can be solved by an extra level of indirection." (Lampson et al. ([135]) attribute this phrase to David Wheeler, citing the authority of Roger Needham).

Metadata is ubiquitous in all computing systems, existing in many forms and for many purposes. Encoding metadata in a machine-readable format such as XML is necessary but not sufficient to enable automated management and use of metadata in an open system (e.g., see

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[144]). It is necessary to have a formal model for the metadata and for the encoding, and mechanisms for managing metadata.

This section presents a general of metadata for Ubiquitous Computing Environments. Section 5.1 proposes a three tier model, which is an indirect mapping between objects of the physical world and concepts of a computational model. The *metadata* provides the intermediate representation for this mapping.

Section 5.2 summarizes the requirements for metadata in this meta-model. A formal language for metadata will be used to address this problem.

5.1. The Meta-model

The overall approach is to develop a model of metadata to act as an intermediate representation between the world of interest (the Ubiquitous Computing Environment) and the computational models to manage and interact with them. Figure 3 suggests the general approach.

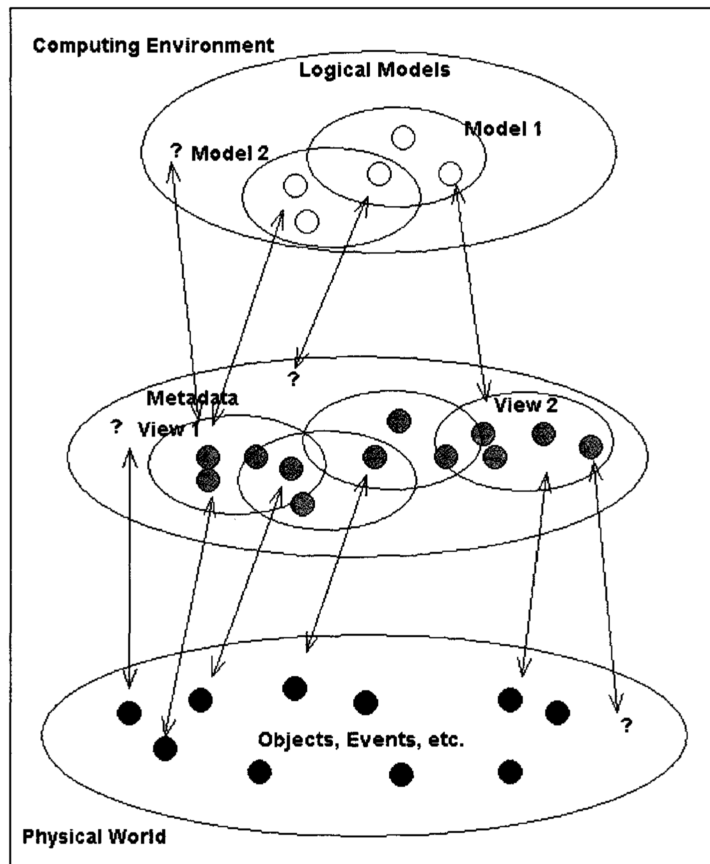


Figure 3. The Meta-model.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The physical world contains objects, events, relations, and other concepts of interest. The Ubiquitous Computing Environment implements one or more computational models of the environment as part of the implementation of applications, services, and infrastructure. These models operate on logical representations of the real world entities. For a given model, the universe of entities and their properties is a subset of the physical world. The multiple models have alternative views of the real world, they may refer to disjoint or overlapping sets of entities, and may define different attributes for the same entity.

The entities of the physical world are represented by *metadata*. Metadata about the physical environment is a set of *statements about the entities*, relations, events, and other concepts. There may be many sets of metadata about the same physical entities, i.e., there may be many different statements about a given physical object.

In this three-tiered framework, the computational model is implemented by mapping the abstract concepts of the model to statements in the metadata. The statements are, in turn, mapped to entities or concepts in the physical environments. Thus, there are two levels of mapping that need to be defined.

5.2. Important Requirements for the Metadata Language

A metadata language must define an abstract language, one or more machine-readable encodings, and operations to prove correctness and other formal properties of statements in the language. This section considers critical requirements for the metadata language.

There are many possible "languages" for metadata, suitable to different purposes, and each language may be encoded in different ways. For a given metadata language, it is usually possible to write many different descriptions of the physical world. In fact, it is usually possible to write many different valid descriptions of the same environment: it is possible to write several statements that are valid within one or more models, yet not equivalent nor even necessarily comparable with each other.

A given metadata language can be interpreted (mapped) into multiple formal models, i.e., the language may have multiple "semantics". So, even if the syntax is fixed (e.g., a set of XML tags), there is no guarantee that two parties will have the same interpretation of the statements. Therefore, it is necessary to have a standard and flexible model for metadata, i.e., a model for defining and interpreting statements about objects. The result is the definition of machine-readable languages that are mapped to formal semantics. Metadata expressed in the language can

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

be parsed into statements with precise logical interpretation, which are therefore amenable to automated reasoning.

Four critical categories of operations are needed. First, the metadata language must be able to be mapped to specific objects, actions, concepts, and other entities. The language constructs be suitable for the purpose, i.e., describing the objects of interest.

Second, the metadata must be kept consistent as the system evolves. It must be possible to verify that statements are valid within the model. This is analogous to the requirements for database schema languages: the metadata must conform to the grammar and specifications of the language and model.

Third, it is necessary to compose and compare instances of metadata from many sources; to cross reference, link together, and/or merge metadata from independent entities. For instance, it is important to be able to combine statements from several sources (e.g., multiple dictionaries) to form a single, correct statement. Obviously, these operations must preserve logical consistency.

Fourth, it is necessary to support queries in various forms. Answering questions requires a design for asking the question, for determining the answer, and for delivering and interpreting the answer. Question answering is particularly difficult in an open, dynamic, system where the questions and answers cannot be known in advance. The metadata language is used to state the query, to interpret the query, and to state the answer.

In the following section, *ontologies* are introduced as a formal language for the metadata of a Ubiquitous Computing System.

6. Ontologies: The Silver Bullet?

Ontologies: a silver bullet for knowledge management and electronic commerce (book title, [56])

In recent years, several lines of research have converged to propose a model of "ontologies" that serve as a lingua franca for heterogeneous information systems [56, 88, 90, 154, 186]. This thesis shows how to apply solutions developed in other domains to Ubiquitous Computing Environments.

The term *ontology* is used in several contexts with somewhat different meanings. Section 6.1 briefly reviews the use of the term ontology in philosophy, Artificial Intelligence, and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Information Science. Section 6.2 shows how ontologies are applied to Ubiquitous Computing Environments. Section 6.3 gives a simple example of how ontologies are used to define metadata.

6.1. Definition of "Ontology"

Philosophers have defined "ontology" as the study of the kinds and structures of all the objects, properties, and relations in every area of reality [215]. In general, a philosophical ontology is the collection of objects that the system (theory) commits to. Importantly, the "ontology" is conceptual, and not a specific representation. These questions were considered by Aristotle more than 2,000 years ago, although the term "ontology" was coined in the seventeenth century.

Scientific disciplines have developed (implicit or explicit) taxonomies or classifications as part of their theory and practice. Some philosophical ontologists have worked to produce ontologies similar to but more general than scientific terminologies (e.g., [192]). Ideally, these ontologies might produce deeper and/or broader understanding and synthesis of the concepts developed in highly specialized sub-disciplines.

Similar efforts have been applied to other organized bodies of knowledge, including legal and business models (e.g., [185]), medicine (e.g., [166]), and folk-science (e.g., [160]). Ontologies have also been used in attempts to support automatic translation of natural languages. In this application, the ontology attempts to be an intermediate language for the underlying concepts expressed in the human languages. In all of these uses, the ontology is intended to be a *theory of the concepts that exist, independent of the specific language(s) used to express them.*

Computer scientists have adopted the term *ontology* for a related, but more concrete concept. Heterogeneous information systems, such as data warehouses or the World Wide Web, face a serious challenge because independent information sources use incompatible vocabularies and conceptual structures. Combining or comparing data from multiple sources (e.g., from one database schema to another, or one Web site to another) is laborious and sometimes impossible (e.g., [17, 56, 74, 98, 186, 193, 218]).

This challenge has been called the "Tower of Babel" problem. Even when possible, pair wise translation between N sources requires N^2 translators, which cannot scale up to large numbers of data sources. In response, researchers have sought to create common vocabularies,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

in the hope that N rather than N^2 translations will be needed. In this work, an *ontology* is *an intermediate language between databases* [186].

Artificial Intelligence research has faced a similar problem among Knowledge Representation systems. Different expert systems use different conceptual organizations and vocabularies, which makes it difficult to integrate knowledge from multiple Knowledge Bases [88]. This problem has been especially critical for Autonomous Agent systems, in which software agents must conduct sophisticated conversations with independently developed agents—"strangers"—that may well have different vocabulary and conceptual schemes. In this field, "ontology" has been used to mean *a shared representation of the entities of a conceptual model* [87, 88, 226].

The Web Services community faces similar challenges. The Web Services Architecture [250] includes the Web Services Description Language (WSDL) [253] and the Universal Description, Discovery, and Integration (UDDI) standards [12]. The former is a standard for describing interfaces, and the latter is a standard for publishing and discovering services. These standards define essential mechanisms for exchanging descriptions of Web Services, but they do not define a data definition or schema language. As a consequence, Web Services from different sources may completely conform to the standard, yet not share the same "vocabulary", and therefore may or may not be able to work together. The so-called "Semantic Web" addresses this problem with a set of standards for expressing and reasoning about ontologies [14, 159, 251].

In summary, philosophers and computer scientists use the term *ontology* to describe different but related concepts [215]. Philosophers would usually take the "ontology" to be purely conceptual, and to be a statement about reality. In computer science, an "ontology" is designed to be a portable intermediate language to represent (describe) the entities of a database or other software system. In general, the computer scientists seek to implement computer-based systems, rather than to model "reality". Furthermore, computer scientists sacrifice power and generality of the theory (ontology) to achieve computational efficiency. This thesis develops and uses ontologies as commonly defined in computer science and the Semantic Web.

6.2. Ontologies in Ubiquitous Computing Environment

This thesis observes that a Ubiquitous Computing Environment faces a "Tower of Babel" problem similar to other information systems: devices, services, and local environments are

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

designed and deployed by independent groups, and in different combinations; yet they must exchange detailed and complex information in order to work together. The exchange of data takes place in message passing and in stored data such as registries, files, or databases. The information of interest is metadata about the entities of the system, as well as about events and other contextual information.

Ontologies provide a general and standard language to address this challenge. Ontologies are a formal "vocabulary" of the metadata, analogous to a database schema: the ontology defines the entities, relationships, and constraints of the system. The ontologies will be integrated into protocols of a local system or context to play the role of a schema for the content of messages. In this approach, statements in the metadata refer to one or more ontologies that define the terminology used.

A formal ontology language assures that the producer of the metadata and any consumers will be able to unambiguously interpret the classifications and constraints specified in the ontology and therefore have the same interpretation of the metadata. An application, service, or agent can use an ontology by retrieving the encoded file (e.g., from a URL), parsing it into data structures (e.g., a graph), and then translating the data structures into logical assertions, proofs, and queries. The formal semantics enables all users to agree on these proofs, and thereby maintain consistency in a decentralized environment.

This study assumes the fundamental principle that *ontologies should be developed by appropriate communities with the requisite domain knowledge*. The domain knowledge includes an understanding of relevant empirical and theoretical understanding, and also of the goals and methods of the users. Communities range in size from whole industries and disciplines, through enterprises and departments, to smaller groups with very specialized interests. It must be possible for different communities to reuse (share) concepts with related domains, develop the ontologies to express the specialized concepts they need, and to use the concepts in a general-purpose infrastructure.

To construct the overall ontology, it is necessary to define shared, standard, high level concepts. At the very apex of the ordering are the most general concepts that apply to all conceptual systems, usually termed an *upper ontology*. Below these are collections of general concepts that describe broad topics.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

From these general concepts mid-level ontologies for broad domains are constructed. These are often structured as hierarchies of general and specialized concepts within a domain. Below these, smaller communities with shared interests may construct specialized ontologies for sub-domains. Thus, there is a conceptual locality to ontologies.

Ontologies are related by reuse of concepts or by defining relationships between concepts. First, an ontology can import concepts from another ontology, which establishes the conceptual relationship between the domains. Second, relationships can be explicitly declared for concepts in different ontologies. For example, two concepts can be declared to be synonyms or one may be declared a subclass of the other. These relations can be used to compose multiple ontologies into a single ontology.

In this approach, the overall ontology (for the whole universe) is a union of ontologies, related as hierarchies of specialization through a graph of relations. The entire ontology is probably too large and complex to exist as a single artifact. However, there is *locality* in the use of ontologies.

The locality has several dimensions: spatial, temporal, and contextual. At a given time and space, a person and program, and local system uses a specific view of part of the world, i.e., a subset of the whole ontology. In other words, for a given context, there is a dynamic local "working set" from the hypothetical universal ontology. These locality principles are developed in Chapter 3.

6.3. Example Use of an Ontology

This section gives a simple example to suggest how an ontology can be used by several applications. The ontology is a classification of entities, along with relations between the classes. Each entity in the system is identified as an instance of one or more class. The ontology can be used to deduce other classes that the entity does or does not belong to.

An application is designed to manipulate entities of particular classes: a given application only deals with the classes that are relevant to its goals. The ontology enables each application to classify entities according to its own requirements, and to deduce a classification when it is not explicitly defined.

For example, consider the trivial ontology for "vehicles" shown in Figure 4. This ontology might be composed from separate ontologies for automobiles, aircraft, and other vehicles, and it would be elaborated to include many specific instances of automobile, aircraft,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

and so on. In addition to the classes, the ontology specifies mandatory and optional attributes of each class and relations between the classes, which are not shown in the diagram.

Entities in the environment are classified into an appropriate class, e.g., a particular object would be defines "class='Toyota Prius', which is a sub-class of "sedan", and so on. The ontology can be used by users and applications with different goals and views of the entities.

For example, suppose this ontology is used in the context of a law enforcement agency to determine which vehicles a particular person may legally operate. The system can use the ontology to discover the concept or concepts that subsume the categories of interest. For example, an ordinary driving license applies to all instances of "automobile", but not to other classes. To discover if a particular type of vehicle requires a driving license, the ontology hierarchy is followed to discover whether it is subsumed by "automobile" or not. Note that two applications can agree on this rule, regardless of the specific sets of automobiles they know about, and regardless of whether they share any other parts of the ontology.

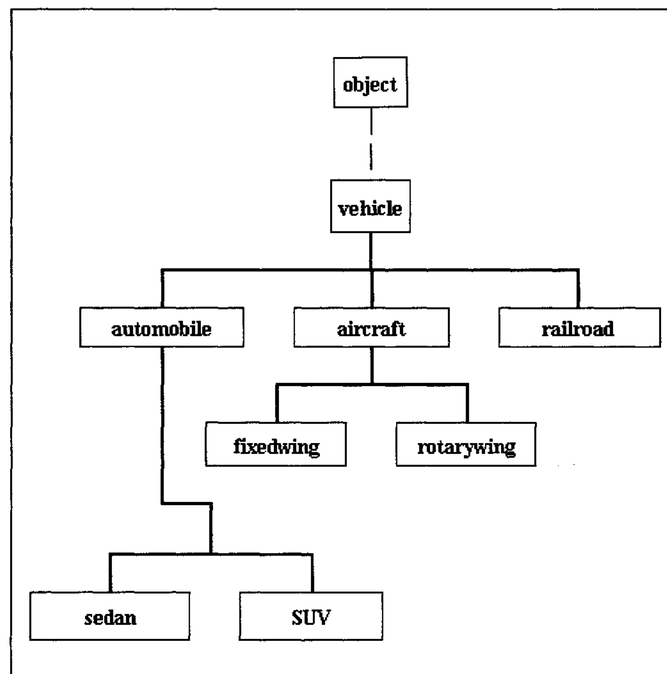


Figure 4. Sketch of a classification of vehicles.

On the other hand, other applications or users might use the ontology in a different way. For example, a regulatory agency might enforce noise pollution regulations that apply to all

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

classes of vehicles; while an insurance company might have different policies for each specific model of automobile. These applications would classify the same objects using different terms from the ontology.

Furthermore, the common ontology enables these applications to exchange information. For example, the insurance company can discover the license and pollution requirements because it can determine that the specific model of interest is subsumed by the class of automobiles and vehicles. Conversely, the pollution control agency can determine that a specific SUV is an instance of vehicle, and therefore subject to regulation. Without the ontology, it would be difficult for these independent systems to correlate their different views of the same object.

7. Related Work

This thesis builds on previous work from several areas. In the following chapters, specific related work is discussed in detail. This section summarizes work related to this thesis that is not discussed elsewhere.

7.1. Standard Interfaces and Services

The semantic infrastructure discussed in this thesis builds on and extends existing standards for managing Knowledge Bases, distributed systems, and services. As stated earlier, distributed object standards such as CORBA [175], .NET [106] or OGSI [232] are the bedrock on which the Ubiquitous Computing Environment must be built. These will be extended with additional services, such as the Gaia environment [205]. This thesis describes an Ontology Service to manage *ontologies*, using one or more Knowledge Bases.

The Open Knowledge Base Connectivity (OKBC) is the standard interface for connecting multiple Knowledge Bases [26]. The OKBC is based on the Knowledge Interchange Format (KIF) ANSI standard [71]. At the heart of the OKBC is a standard for assertions (*tell*) and queries (*ask*) to a KB. These operations are simple interfaces that pass messages containing KIF sentences: the OKBC is essentially an interface for exchanging KIF. When this standard is updated to support the Semantic Web standards, it will be the ideal standard for the back end of the Ontology Service.

Siming Chen et al. describe an Ontology Service within a Grid-based Knowledge Engineering environment [35]. This system is implemented with DAML+OIL and the FaCT server, analogous to the prototype described in this thesis. Other Grid services would use the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Grid Ontology Service to implement services of the so-called "Semantic Grid" [34, 75]. Many other Knowledge Engineering environments have similar services [2, 19, 44, 57, 58, 154, 155, 159, 189, 218]. These systems are primarily designed to assist searches across multiple data sources. In contrast, this thesis considers a service for the computing infrastructure, i.e., in protocols between system components rather than human users.

7.2. Discovery, Matchmaking, and Brokering

In recent years, several so-called "discovery" protocols have been introduced, with the overall goal of making digital networks easier to create and use [144]. "Discovery" is used to refer to a more spontaneous process, in which entities "discover" the other entities on the network, and present themselves to other entities. The most important features of a discovery protocol are:

- "Spontaneous" discovery and configuration of network devices and service.
- Selection of specific types of service
- Low (preferably no) human administrative requirements
- Automatically adaptation to mobile and sporadic availability
- Interoperability across manufacturers and platforms

The classic Internet protocols, such as the Internet Domain Name Service (DNS) [165], do not meet these requirements because:

- They use static databases/files of information
- They are required to be maintained by privileged administrators
- They do not guarantee the availability of the objects registered
- They have limited semantics for searching
- They do not generate events when resources register and unregister

Some directory services such as LDAP [255] and CORBA Name and Trader Services [175] can be used for service announcement and requests, but do not themselves specify protocols for spontaneous discovery. These services define interfaces, protocols, and languages for advertising and look up, but do not define the contents of the metadata, or mechanisms for defining grammars.

For example, the CORBA Trading Service is a standard interface for a broker, which defines a language for advertising and query, but (naturally) does not define the properties of the advertised services, or the legal values of properties. By design, the specification of valid

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

properties and relationships is left to communities, such as the CORBA Domain Task Forces [174]. Ontologies are a natural means to specify, publish, and manage these metadata languages.

In earlier work, several discovery protocols were reviewed and analyzed [144]. This work predated the emergence of the Semantic Web, which is intended to overlay or replace these older technologies

The Salutation protocol is an open specification that provides "spontaneous" configuration of network devices and services. Salutation is in use by a consortium of companies that make printers and similar devices, including HP, IBM, Xerox, and AOL [187]. Salutation is also designed for and highly compatible with wireless technologies, there are already Salutation bindings for IrDA and Bluetooth.

One interesting feature of Salutation is that it defines a specific (extensible) record format for describing and locating services. This format includes service type (such as '[PRINT]') and attributes (such as 'color'). Clients can query for services by standard attributes, and the registry returns the address and a "Personality Profile", which is a description of the service and its interface. It is worth noting that the profiles are specified in great detail (e.g. the profiles for printers and similar devices ([209]) is 275 pages long!). The record format and profiles are exactly the sorts of models and metadata that is needed to define Ontologies for these devices.

The Service Location Protocol (SLP) comes from Sun Microsystems, and is an IETF standard for "spontaneous" discovery of services [91, 92]. SLP defines an abstract architecture consisting of "User Agents" (UA) (clients), "Service Agents" (SA) (services) and "Directory Agents" (DA) (directories). SLP is designed to work well with LDAP, but does not require LDAP.

The SLP defines a "Service URL", which encodes the address, type, and attributes of the Service [91]. Attribute matching is specified by a template and an LDAPv3 predicate [92]. The SLP does not define the content of the URL or attributes. Standard ontologies fill an important gap in this protocol: Ontologies could be used as a grammar to generate and check the SLP Service URLs.

The JINI Discovery Service resembles SLP, and was clearly influenced by it. However, JINI is very tightly bound to the Java environment. The protocol is mostly defined as exchanges of serialized Java objects, mostly via Java Remote Method Invocation (RMI) [53].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Servers advertise by registering a Java RMI stub with the JINI Lookup Service. Clients locate services by requesting specific types of service. The request is basically a simple template for matching string attributes. However, the request and the matching must be implemented as Java objects, following JINI specified interfaces. Ontologies could be used as a grammar to generate and check JINI service offers and queries.

Universal Plug and Play (UPnP) is a Microsoft standard for spontaneous configuration [37, 162, 163]. UPnP handles network address resolution, and coupled with the IETF proposal Simple Service Discovery Protocol (SSDP) [55] it provides higher level service discovery. UPnP uses XML for device/service description and queries.

Services are described by extended URLs, similar to (but completely incompatible with) SLP. The URL is for an XML file with an elaborate description of the device. Starting with this URL, the SSDP defines a Web based discovery protocol, which uses HTTP (with extensions).

A UPnP "device" is said to export one or more "services". Services are describe in XML, and the XML can be a complete abstract description of the type of service, the interface to a specific instance of the service, and even the on-going (virtual) state of the service. The interface and state descriptions are intended to allow clients to implement custom interfaces to devices, by mapping local displays and operations to the abstract state and interface represented in the XML. The XML description can be used by programs or browsers to locate specific services by filtering on XML tags or combination of tags. XML is extremely flexible, so it can deliver almost any kind of information. These descriptions go far beyond the information available from SLP/LDAP, Salutation, or JINI. It would be interesting to use ontologies to define the vocabulary for these XML messages.

The Secure Service Discovery Service (SSDS) is part of the University of California, Berkeley Ninja research project [40, 83]. The SSDS is similar to other discovery protocols, with a number of specific improvements in reliability, scalability, and security. Although SSDS is implemented in and relies on Java, it uses XML for service description and location, rather than Java objects.

The protocol is designed as an exchange of XML "documents", and service location is done by matching of XML tags. This is logically equivalent to UPnP, and the two flavors of XML should theoretically interoperate by automatic mapping. It is argued that other forms of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

service advertisements, including JINI objects, can be translated into XML as well ([40], p. 33). Again, ontologies could be used to define a grammar for these XML messages.

Discovery is a significant problem for agent-based systems as well. Different expert systems use different conceptual organizations and vocabularies, which makes it difficult to integrate knowledge from multiple Knowledge Bases [88]. This problem has been especially critical for Autonomous Agent systems, in which software agents must conduct sophisticated conversations with independently developed agents—"strangers"—that may well have different vocabulary and conceptual schemes [87, 129, 226].

Querying a Knowledge Base is a complex and subtle problem. It is necessary to a formal description of the semantic relationship between the query, the query answer, and the KB(s) used to provide the answer (e.g., see [61, 62]). Providing adequate results is a significant challenge, because explaining automated reasoning is an extremely difficult problem [254]. For example, McGuinness gives a definitive study of the problem in the context of Description Logic [153, 157]. This thesis does not address this deep and difficult problem.

Information systems such as multiple database systems, the World Wide Web, and the Grid face similar challenges [2, 19, 34, 35, 58, 139, 159, 189, 196, 210, 218]. The Web Services Architecture [250] has emerged as a set of standards, including the Universal Description, Discovery, and Integration (UDDI) standard for publishing, discovering, and composing independent services in an open network [235]. The UDDI has limitations similar to the registries discussed above. In fact, the UDDI standard does not even define a data definition or schema language. In order to meet the limitations of UDDI, the "Semantic Web" provides standards for expressing and reasoning about vocabularies and relationships [14, 59, 159, 170, 251, 261]. This technology is considered in detail in later chapters.

7.3. Ontologies in Ubiquitous Computing Environments

Ontologies can be used in the design and configuration of systems and software. The formal taxonomy of an ontology can help develop correct and consistent database schema and abstract class graphs (e.g., [186, 217]), implement queries and caching between distributed databases (e.g., [78, 79, 161]), implement correct message passing protocols (e.g., [87, 124]), and reasoning about proper combinations of components and interfaces (e.g., [6, 158, 168]). Ontologies may be useful for flexible Human Computer Interfaces (e.g., [29, 38, 230]). In this

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

application, ontologies can help select "semantically" appropriate interfaces (e.g., [5, 223]), and can be used to explain the interface to users (e.g., [157, 197]).

Automated reasoning and learning have been applied to attempt to automatically discover trans-ontology relationships between large ontologies. For example, the OntoMerge project promises to merge any two DAML+OIL ontologies using automated learning to discover bridging axioms [52]. Doan et al. use statistical text processing algorithms to attempt to discover trans-ontology relationships [50, 51], and others have attempted similar learning approaches (e.g., [11]). These approaches depend on the existence of large bodies of data (e.g., Web pages). Where these techniques can be applied, they may yield "bridge rules" to enable "translation" between vocabularies.

Several current research projects are incorporating Web Ontology Language (OWL) [249] or similar XML ontologies into context-aware, Ubiquitous, and Pervasive Computing Environments. These projects show an emerging need for both community-based ontologies for Ubiquitous Computing, and for a common architecture for managing ontologies in local environments.

For example, the Context Broker Architecture (CoBrA) is an infrastructure for context-aware computing that includes a service to manage a Knowledge Base constructed from OWL XML Ontologies. Ontologies classify and define entities important for Pervasive and Ubiquitous Computing, including devices, users, and events. The Ontology Service is used to implement an enhanced look up service (termed "context-sensitive resource discovery) [27].

In another example, the STEER environment ontologies provide common abstract descriptions of Web Services, which can be used by scripts to dynamically bind to appropriate services in the local environment [143]. In a similar vein, a Grid Ontology Service [35] would be used to implement services of the so-called "Semantic Grid" [34, 75].

Several other projects have proposed similar uses of ontologies (e.g., [5, 38, 150, 229, 230]). Together, this work represents an emerging consensus that has led to recognition of the need a standard infrastructure for managing ontologies.

This thesis is principally concerned with managing the system. The infrastructure discussed here can be used in protocols for advertising and discovery, message passing, and service composition. A semantic infrastructure requires mechanisms for managing multiple ontologies, especially, a mechanism to compose ontologies from multiple sources. Composing

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

two arbitrary ontologies is essentially the same problem as automatically integrating or matching two database schemas. The formal complexity of this problem is not known, but no general solution is known, and some very similar problems are known to be undecidable [17, 74, 98, 193].

The general practice is to conduct a manual cross-walk, analyzing the ontologies to discover trans-ontology relationships (e.g., [56, 186, 218]). This task is aided by tools such as OilEd [8] or OntoEdit [181, 221] and research environments (e.g. [57, 156]). These tools help visualize the ontologies and can store and validate proposed mappings. These tools generally produce a single merged ontology.

Borgida and Serafini propose a distributed approach, in which multiple separate ontologies are related by "bridge rules" [17]. Others have proposed similar approaches in the domain of multiple database schemes (e.g., [74, 97, 98, 193, 227].) These approaches require a cross-walk, but do not require a global, merged ontology.

Ontologies are an important foundation for intelligent agents and brokers that seek to automatically compose services (e.g., [87, 168, 226]). The ontologies provide a mechanism to discover and reason about the entities available in the current system in order to apply higher-level rules. In the future, ontologies will be integrated into higher-level languages that express and reason about rules (e.g., [6, 85, 86, 114, 216]).

A standard semantic infrastructure will be shared by these and other advanced services. The Ontology Service described in this thesis is an important foundation to enable the construction of Ubiquitous Computing Environments.

7.4. Requirements for Ubiquitous Computing Environments

This thesis adapted and extended previous work from several areas of research to address important problems for a Ubiquitous Computing Environment (UCE). The developments presented in later chapters build on related work discussed in the previous section. This earlier work was the necessary foundation, from which this thesis developed theoretical and practical techniques needed for UCE.

A Ubiquitous Computing Environment is a local environment in which a very dynamic and heterogeneous set of activities must be supported. In addition, the UCE must manage the physical environment, as well as software and devices. The UCE must be a seamless: mobile

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

users and objects should be able to enter and leave a local environment without user or administrative intervention

The UCE is also characterized by a locality of use, i.e., a local space needs a dynamic "working set" from a larger universe of components and activities. This key insight recasts the way metadata is used, and led to the developments presented in this thesis.

This thesis considers one of the basic problems for any decentralized system, resource discovery. Resource discovery is a key problem for all decentralized systems, including digital libraries [147, 210], intelligent agent systems [87, 129, 168, 226], and the World Wide Web [14, 58, 159, 170, 262]. In all these applications, there are multiple, independent entities that must exchange messages but do not share a single vocabulary or database schema. This thesis observes that UCEs face a similar challenge, and therefore adopts developments from these applications areas.

In a distributed system, resource discovery is done as part of registration, advertising, look up, and event services. In a UCE, these protocols must operate seamlessly and in real time, and should provide a localized view of the dynamic local environment. In order to operate seamlessly, it is necessary to replace manual tasks with automated algorithms as much as possible.

Ontologies have been developed as a language for managing metadata in distributed information systems. Ontologies and the underlying formal logic are extremely general and powerful, so it is no surprise that they can be applied in UCEs. Description Logic, Knowledge Bases, and the Semantic Web XML languages provide the necessary foundation, which must be extended to work well in the UCE.

Ontologies are used to manage schemas and data in information systems and the World Wide Web. These systems focus on organizing and managing complex bodies of data. The ontologies are created manually; to construct a single, large, slowly changing, schema for the system.

Fusing and matching data from multiple sources is a critical problem for these applications, ant this requires matching across multiple schemas (or, more abstractly, multiple ontologies). The general practice is to conduct a manual cross-walk, analyzing the ontologies to discover trans-ontology relationships. The formal complexity of this problem is not known, but

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

no general solution is known, and some very similar problems are known to be undecidable [17, 74, 98, 193].

For a Ubiquitous Computing Environment the central challenge is diversity and real time change, rather than scale and complexity of the information to be managed. Furthermore, the system must operate seamlessly, with little human intervention, so manual cross walks are not feasible. This thesis presents principles and techniques to address these challenges.

Chapter 3 proposes locality principles, which suggest that ontologies for UCE should be developed piecemeal. The ontology languages discussed in this thesis are well suited for encoding many small ontologies. Second, this thesis proposed that each local space should use a dynamically constructed "working set" from an overall pool of ontologies. While familiar from other contexts such as virtual memory systems, this principle has not been used for ontologies before. In order to achieve this, it was necessary to develop a method for automatically composing multiple ontologies with minimal human intervention.

Broker- and agent- based systems use ontologies to improve discovery of that fill certain goals, i.e., given a user request, automatically locate and execute services to meet the goal. This thesis builds on previous work that has developed *matchmaking*, and so called *discovery protocols*. If the UCE is conceived as an environment in which many brokers and agents will operate, it is clear that a common metadata language is needed, along with common services to manage and query using the metadata.

This thesis builds on efforts that focused on heuristics to use proofs in Description Logic to define a set of concepts that match a given query concept. The previous work was synthesized and extended to create queries that seem to better meet the queries of a UCE.

Most Pervasive and Ubiquitous Computing Environments have ad hoc metadata, with little or no use of formal languages and reasoning. It is now widely recognized that semantic services will be needed. This thesis has showed the need for ontologies in Ubiquitous Computing, and developed theory and key algorithms to make this possible.

The following chapters review earlier work in more detail in the appropriate contexts. Chapter 8 summarizes the relationship of earlier work, to highlight on the distinctive contributions of this thesis.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

8. Summary

This chapter presented basic definitions and assumptions about the Ubiquitous Computing Environment. The environment is characterized by a diverse set of components, environments, users, and applications.

The Ubiquitous Computing Environment must map the virtual world (computational models) with the physical world. The metadata of the system provides a level of indirection between the computational models and the entities of the environment. The metadata enables flexible bindings between virtual and physical objects, and also allows mapping between multiple virtual models, e.g., different views of the same physical objects. This thesis develops semantic infrastructure, to manage and query ontologies in a Ubiquitous Computing Environment.

This chapter presented a three tiered model for the system, and argued that there must be a formal model for the metadata of the system. *Ontologies* were introduced as a formal language for metadata.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 3. The Model and Approach

1. Introduction

Chapter 2 introduced the problem of managing metadata in a Ubiquitous Computing Environment. In the Ubiquitous Computing Environment, abstract models are mapped to the physical world. The metadata of the system provides a level of indirection, to make this mapping flexible and dynamic. Formal ontologies are a language to express the "vocabulary" of the metadata.

This chapter presents an approach to solve key problems in creating, managing, and using ontologies in a Ubiquitous Computing Environment. This chapter reviews theoretical foundations and Chapter 4 develops algorithms to manage ontologies. Together, these developments are the necessary foundation for the Ontology Service.

Section 2 presents the overall approach. The section discusses how the concepts of a model should be captured in one or more ontologies, and encoded in a formal language. An Ontology Service is proposed, which is the essential component of a semantic infrastructure. The Ontology Service manages one or more context-specific ontologies for a local environment. The Ontology Service uses a Knowledge Base to implement proofs and queries using ontologies.

Section 3 presents a fundamental design pattern that is the core of models for Ubiquitous Computing Environments.

Section 4 presents a formal language for metadata, based on Description Logic. Description Logic provides a language for encoding ontologies and proving logical consistency. Description Logic also supports logical queries. Section 5 shows how the proofs of Description Logic are applied to prove the validity of an ontology. These proofs are the foundation on which the Ontology Service builds its services.

Chapter 4 and Chapter 5 show how concepts from these models can be encoded in ontologies, which can be managed and used via the Ontology Service. A prototype implementation is presented in the following chapters.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2. Overview: Ontologies for Ubiquitous Computing Environments

Chapter 2 introduced *ontologies* as a language for metadata. Specifically, ontologies represent a formal "vocabulary" of the metadata, analogous to a database scheme: the ontology defines the entities, relationships, and constraints of the system. These ontologies are managed by an Ontology Service. The Ontology Service implements protocols and algorithms for managing and querying ontologies.

In order to use ontologies in a Ubiquitous Computing Environment, several challenges must be met. This section shows how to solve these problems, in order to create a new "semantic infrastructure." The general approach is for the concepts of abstract models to be encoded in one or more ontologies.

2.1. Overview of the Use of Ontologies

As discussed in Chapter 2, ontologies are created from models. The concepts of a model are represented in the logical statements of the formal ontology. Ontologies are very flexible; they can represent concepts and relationships from many sources, including natural classifications, informal models, or ad hoc collections.

The ontology is encoded in a formal language, i.e., a language with a specification in formal logic. The formal semantics can be used to prove logical consistency, deduce relations, and answer queries. The formal proofs and queries reflect important properties of the vocabulary, consistency, and queries about the relationships and constraints.

In a distributed system, the encoded ontology is used by producers and consumers. An application, service, or agent can use an ontology by retrieving the encoded file (e.g., from a URL), parsing it into data structures (e.g., a graph), and then translating the data structures into logical assertions, proofs, and queries. The formal semantics enables all users to agree on these proofs, and to maintain consistency in a decentralized environment.

Figure 5 summarizes the conceptual approach. The ontology (2) is a formal representation of the concepts of a model or other specification (1). The ontology is encoded in a formal language (3). Even if the model itself is informal or ad hoc, its concepts can be formalized in an ontology and used in the system.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

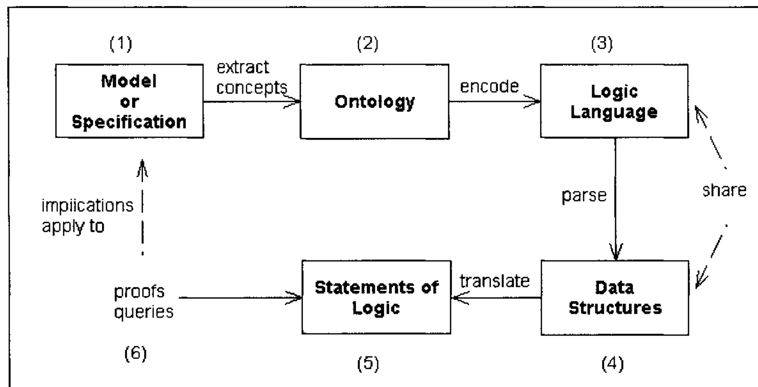


Figure 5. Use of Ontologies.

The ontology can be shared among different applications, services, and systems. Each user parses the encoded ontology to create data structures (4). Different data structures might be used by different implementations; typically, the ontology is represented as a labeled graph. The data structures are interpreted to generate statements in logic (5). Proofs (or queries) on the logic imply facts about the ontology and the original model (6).

In a Ubiquitous Computing Environment, ontologies are used to define a grammar for the metadata of a local system or context. Statements in the metadata refer to one or more ontologies that define the terminology used. The formal ontologies enable producers and consumers to agree on terminology and to translate when equivalences are known. Ontologies are integrated into protocols, to define the content of messages.

Figure 6 shows a simple example of some metadata describing the context of a Ubiquitous Computing Environment; the activity of two actors. The metadata refers to the ontologies that define the concepts, e.g. the concept 'Person', is defined in ontology 1, which might be represented in a statement such as: 'Bob isA ontology1:Person'.

Note that the ontologies are used to define the categories of actors and actions, but not the properties of the specific actors. That is, the current ontology defines the properties of "cat", but not the properties of any specific cat.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

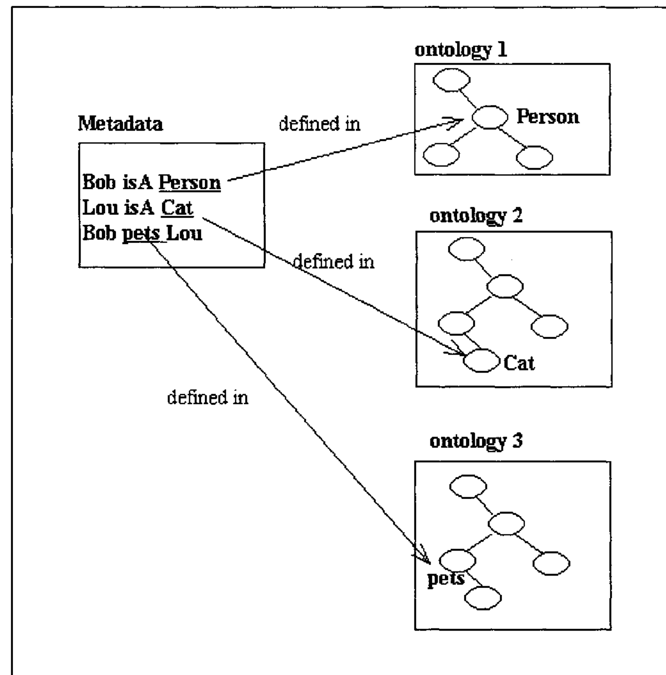


Figure 6. Metadata refers to one or more ontologies.

2.2. A Hybrid Model for Service Registration and Discovery

This section proposes a model architecture for how the protocols of the infrastructure will be augmented with semantic information. The model is a hybrid system: semantic information is managed by the Ontology Service, while other services manage the state of the system.

A registry is database of service descriptions (advertisements), with an interface and protocols to update and retrieve records. A registry represents the current state of the system, i.e., the entities that are currently available in the system. The Ontology Service augments these services, providing a common vocabulary for the protocols. Figure 7 shows a sketch of this architecture.

This section considers two key protocols, *registration (advertising)* and *look up (discovery)*. Service *registration* is the process by which entities are introduced to a local environment. There are many variations of this basic operation, but they all have the same function: to update the state of the system when an entity enters (or leaves). A *discovery* or *look up* protocol seeks to find entities available in a specific environment at run time. As the state of the system evolves, the registration protocol implements updates, and the discovery protocol retrieves information from the updated state.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

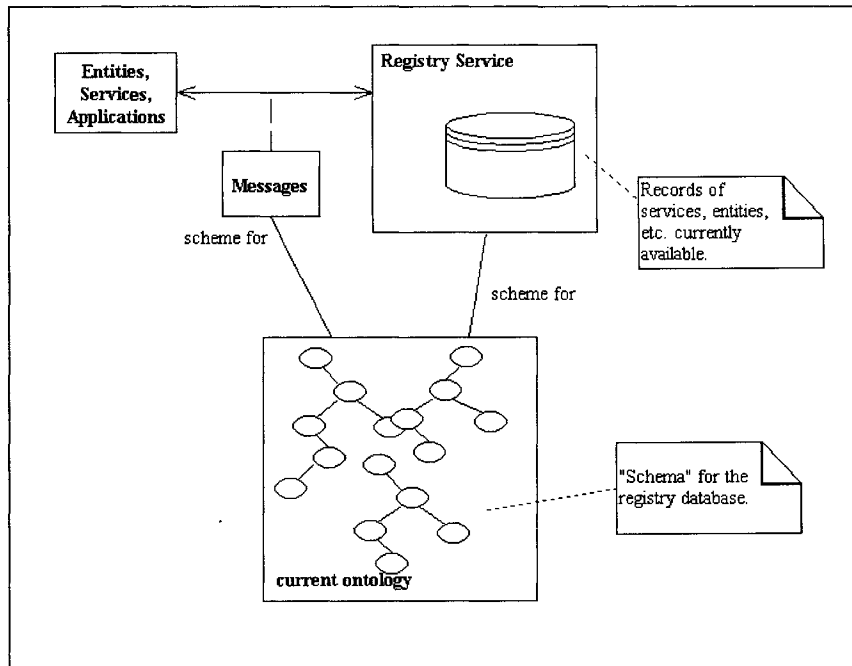


Figure 7. A hybrid design for registry services augmented with ontologies.

This section outlines a model for how to use ontologies to enhance these protocols. Section 2.2.1 describes the contents of the Knowledge Base and the registries. Section 3.1.2 shows how ontologies are used to augment the registration protocol by defining the schema for the records in the registry. Section 2.2.3 shows how ontologies are used to implement improved queries, i.e., *semantic queries*.

2.2.1. Contents of the Knowledge Base: Service Class versus Service Instance

The discovery or matchmaking process uses two related kinds of knowledge about the system: classifications (analogous to the scheme for a database; what *can* be in the system) and state (analogous to the records of a database; what *currently is* in the system). The classification defines a logical model of what entities and relations must and may occur, along with constraints on their properties. Ontologies represent important aspects of the classifications defined by a model.

The running system is populated by entities (people, places, devices, software components, etc.). Each entity is modeled as an *instance* of one or more classes of the ontology. As the system runs, the set of instances constantly changes as objects are created and deleted. There may be zero, one, or many instances of a given class available at a given time. In contrast,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the classification changes only when new classes of entity are introduced, or existing classes modified or deleted, and each class is unique.

A user, agent, or service queries about the current state of the local system. The query may ask about classes or about instances. In the former case, the result must reflect the current set of classes that *may* be in the system. In the latter case, the result set must reflect the objects that *are* available in the system.

In the query and answer process, there is an important distinction between queries about classes and queries about instances. The former asks about "what kinds of entities match this request", which is answered based on the logical design of the system. The latter asks about "what entities match this request (at this time and place)", which is answered based on the state of the system. As entities enter and leave the environment, the former question should give the same answer (the classes of entities have not changed), while the latter should give a completely different answer (the set of available entities).

Many Knowledge-Based Environments are designed to manage both class and instance information. In such a design, the Knowledge Base has a complete view of the system, and uses the same mechanisms for assertions and queries about both classes and instances. There are two important challenges for this approach.

First, the Knowledge Base (KB) must be kept consistent with the current state of the system, i.e., with the state of all services and objects in the system. If the KB and the state of the system become inconsistent, then the automated reasoning of the KB is rendered useless. This requires that any action that changes the system (e.g., creation of a new object, the departure of an object) must update the KB.

Furthermore, the KB must be kept current in *real time*. If the system is very dynamic, with many entities and objects entering, leaving, and changing, then the KB will need to be updated rapidly and continuously. The KB will need to be able to add, delete, and modify instances very efficiently.

Second, the state of the system must be kept consistent with the state of the KB, including all logical constraints and dependencies. For example, if the KB deduces a logical inconsistency between two services, it must contact the services to resolve the conflict. This might require the KB to have a complete model of the state of all services, and some mechanism for the KB to check and modify the state and behavior of any entities in the system.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The Ontology Service could potentially manage a Knowledge Base of both the classes and the instances. However, in light of the challenges discussed here, this may not be desirable or feasible. For this reason, the proposed architecture is a hybrid: the Ontology Service manages information about classes, while other services manage the instances (Figure 7). This approach decomposes the overall problem, and uses different mechanisms to solve the different challenges. Borgida and Brachman proposes a similar idea for interfacing a Knowledge Base to an SQL database [16].

2.2.2. Service Registration

There are many examples of registries, which are implemented in different ways (e.g., UDDI [235], LDAP [255], the CORBA Naming Service [175], CORBA Trading Service [176], Globus Resource Specification [73], or the JINI Discovery Service [219]). A registry service is implemented with files or a database. Each implementation has a specific scheme and storage format for its records.

A service registry protocol defines messages to advertise a service of entity. The advertisement is essentially a description of the service or services offered, along with information necessary to obtain the service, such as addresses and interfaces to use. The service registration protocol can be augmented to include automated registration of metadata in the Ontology Service. The registration protocol would be extended with the following steps:

1. When a service registers with the environment, in addition to the standard information (name, address, software interfaces, etc.), the service specifies one or more ontologies that are needed to describe the service.
2. The service describes itself by sending a message to the registry, referring to terms in the ontologies specified in step 1.
3. If the ontologies specified in step 1 are unknown to the system ontology, the registry updates the system ontology. Chapter 4 describes a key algorithm to compose two ontologies.
4. The registry registers the service as an instance of the class described, and extracts additional metadata from the description.

The entity uses ontologies to defines the "vocabulary and grammar" of its service description. The registry and other services use the current ontology to define their own "vocabulary". The

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

current ontology enables the parties to discover equivalent and related terms (assuming the terminologies are compatible).

2.2.3. Discovery: Semantic Queries

A query protocol defines an interface and language for queries and results, and also defines the algorithm for answering the query, i.e., how an answer is created. In the hybrid design, the overall query answering process is decomposed into three phases. First, the set of classes that matches the query is discovered. Second, the instances of all the classes in the result set are discovered. Third, the instances are filtered to match additional constraints of the query, such as specific values or ranges for attributes. Table 1 lists these steps and their results.

The first step is the "semantic query": the system deduces all answers that "conceptually" match the query. The subsequent steps are conventional retrievals, using the result of the first phase as part of the query, e.g., "find all instances of these classes". From the point of view of a database, the semantic query implements a form of query refinement and/or expansion.

Table 1. The hybrid model for semantic queries.

Step	Implemented by	Result
1. Discover all classes logically consistent with the query	Query to the Ontology Service and KB	The categories that could answer the request
2. Discover all instances of the classes above	Queries to one or more system services	The candidates that could answer the request
3. Filter result	Application specific rules or agent	The candidates that best match the request

The result set is processed by an application specific set of criteria. In some cases, the query may be specific enough that there is only one result. But many queries will return a set of results that may be ranked in different orders depending on user or application preferences. This must be implemented by the query agent.

3. Methodology for Developing Ontologies: Divide and Conquer

While ontologies can serve as a flexible mechanism for defining and using metadata from many sources, it will be necessary for diverse individuals and organizations to develop ontologies that can be used together with minimal human intervention. Given the variety and rapid evolution of Ubiquitous Computing Environments, creating these ontologies is a challenge.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Designing an ontology for the concepts of a model is a *knowledge acquisition* problem, which is inherently difficult. Fortunately, ontologies can be defined for limited, tractable domains. In a Ubiquitous Computing Environment, there is also a locality of use, because a local environment only needs a small subset of the total universe of metadata concepts. These locality principles lead to a natural divide and conquer method for constructing ontologies.

To construct the overall ontology, it is necessary to define shared, standard, high level concepts. At the very apex of the ordering are the most general concepts that apply to all conceptual systems, usually termed an *upper ontology*, capable of expressing all concepts of interest to all domains and sub-domains.

From these general concepts, mid-level ontologies for broad domains are constructed. These are often structured as hierarchies of general and specialized concepts within a domain. Below these, smaller communities with shared interests may construct specialized ontologies for sub-domains.

Looking from the top down, to construct an ontology it is necessary to define shared, standard, high level concepts. At the very apex of the ordering are the most general concepts that apply to all conceptual systems, usually termed an *upper ontology*. Below these are collections of general concepts that describe broad topics. From these general concepts, mid-level ontologies for broad domains are constructed. These mid-level ontologies are often structured as hierarchies of general and specialized concepts within a domain. Smaller communities with shared interests may construct specialized ontologies for sub-domains, and so on.

An individual ontology is related to other ontologies by reuse of concepts or by defining relationships between concepts. The ontologies need not form a simple hierarchy; in principle an ontology can refer to concepts from any other ontology (subject to the logical constraints of the ontologies). However, as in many other design fields, a hierarchical organization makes it easier to define ontologies. When the ontologies are hierarchically organized, domain ontologies form a natural unit of sharing and reuse, i.e., to "import" a whole domain ontology as a package.

The following sections present these locality principles, and show how these locality principles can be applied to create a hierarchy of related ontologies for Ubiquitous Computing Environments.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.1. Locality: a Fundamental Design Principle for Ontologies

The design and use of ontologies exhibits *locality* in several dimensions: *conceptual*, *spatial*, *temporal*, and *contextual*. Conceptual locality means that a large ontology can be constructed piecewise. The *spatial*, *temporal*, and *contextual* locality means that ontologies may be used piecewise.

Capturing the concepts of abstract models in one or more ontologies is a *knowledge acquisition* problem, which is inherently very difficult and labor intensive [127]. Conceptually, the overall ontology (for the whole universe) is a union of ontologies, related through a graph of relations. This hypothetical ontology is probably too large and complex to exist as a single artifact. Fortunately, there is a *conceptual locality* in ontologies: ontologies can be defined for limited, tractable domains they can be created piecewise.

At a given time and space, particular activities are executed. In this local context, the local system needs only a limited view of part of the world. This means that the ontology for a local time and place is a subset of the ontology for all times and spaces: there is a *spatial*, *temporal*, and *contextual* (task specific) locality in the *use* of ontologies. This locality of use enables a specific application or local environment to dynamically construct the local "working set" from the hypothetical universal ontology.

3.2. A Hierarchy of Domain Ontologies

Applying the conceptual locality principle, ontologies are best developed by appropriate communities with the requisite *domain knowledge*. The domain knowledge includes an understanding of relevant empirical and theoretical issues, and also the goals and methods of the users. Communities range in size from whole industries and disciplines, through enterprises and departments, to small groups with very specialized interests. Of course, communities should to reuse (share) concepts from related domains, to develop the ontologies to express the specialized concepts they need.

This process is an extension of the practices of standards bodies and organizations. *Domain experts* and standards bodies define concepts and develop the formal vocabulary for domains that are published as standards documents and reference implementations. An important goal of an ontology is to formalize this process, to generate a formal specification of the domain-specific vocabulary, and to provide a universal mechanism for disseminating the standard.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The development of ontologies builds on knowledge from many sources, not just formal standards. An ontology might be developed from:

- a theoretical model,
- software class hierarchies (e.g., [152, 203]),
- "metadata standards", i.e., grammars specifically developed for classification (e.g., [177]),
- protocols and policies (e.g., [124]),
- human experts, (e.g., medical classification schemes such as Unified Medical Language System (UMLS) [166]), or
- business models and procedures (e.g., [185]).

Whatever the source of the knowledge, the ontology must define the set of concepts and relations that constitute the "vocabulary" of the domain. This process requires judgment, because there are usually many views of the same concepts.

3.3. Upper Ontologies

An ontology inevitably includes very general ("high level") concepts, along with increasingly detailed and specialized concepts. There is a natural trade-off across these levels of detail: the most general concepts may be widely applicable, while specialized concepts are less widely useful, but are necessary for a specific task. Furthermore, different viewpoints slice up the world in different ways, leading to alternative conceptual models and incompatible low-level vocabularies. These trade-offs are a normal and natural consequence of the process of classification and the organization of knowledge.

It is important to develop ontologies that share an appropriately universal set of high-level concepts, yet also express the needed specialized concepts. In principle, there should be an all-encompassing *upper ontology*, capable of expressing all concepts of interest to all domains and sub-domains. Examples of upper ontologies include Basic Formal Ontology (BFO) [15] and the IEEE P1600.1 Standard Upper Merged Ontology (SUMO) [121].

An upper ontology tends to be extremely abstract, and therefore quite distant from actual use in computer applications. For this reason, it is generally possible to define and use lower level, application- and community-specific ontologies without reference to a comprehensive upper ontology. For example, concepts such as "Device" and "Person" can be taken as given, without reference to more general concepts, such as "Machine" or "Animal".

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Ignoring the upper ontology sacrifices power for simplicity, with some significant risks. First, the ontologies may well suffer from conceptual confusion and deficiencies, stemming from poorly selected concepts or faulty definitions. Second, different ontologies may not be translatable, since they lack a true common conceptual framework [186]. On the other hand, for many purposes the upper ontology is almost irrelevant. Users and applications generally use the specialized set of concepts that make sense for their tasks, which is the domain or sub-domain ontology. The upper ontology is usually needed only to assure universal interoperability and correctness, which is required relatively rarely.[3]

3.4. Locality of Use: A Local Working Set

Unlike the World Wide Web, a Ubiquitous Computing Environment is a local environment. An ontology of the concepts needed for a local time and place is a subset of the (hypothetical) ontology of all times and spaces. As the system evolves, entities enter and leave and local services are reconfigured.

As the system evolves, the "meaning" of the metadata must evolve as well, so that generic messages should be automatically given context-specific meaning. The same message (e.g., a query) may have a different interpretation each time it is sent (e.g., the answer to the query may be different) because the system configuration has changed. The system can implement this evolution by updating the local ontology with new concepts, relations, or constraints.

An important insight is that a Ubiquitous Computing Environment should maintain a local *working set* of concepts from the universe of all ontologies. Maintaining such a local working set requires a mechanism to update an ontology in real time with minimal human intervention. The local infrastructure needs mechanisms to automatically load, validate, and compose ontologies. The rest of this paper shows that ontologies the Semantic Web are well suited for this task. Section 5 presents the formal underpinnings, and Section 6 sketches how this can be used to implement this approach.

---

[3] However, certain applications have stringent requirements for universal correctness, such as financial transactions and security protocols. For these applications, it is critical to base ontologies in a sound upper ontology.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4. A Fundamental Design Pattern: A Specialization of the "Proxy" Pattern

Distributed object systems are based on several fundamental design patterns, especially the "Proxy" pattern and related patterns such as "RemoteProxy", "Adapter", and so on [76, 120]. A Ubiquitous Computing Environment can be said to apply these patterns to objects that are not software entities, including *objects that may have no computing capability at all*, such as people or books.

This pattern is widely (if unsystematically) practiced. In some systems, proxies are explicitly used in this way, as when people are represented by "User" objects. In other systems, this pattern is used implicitly, as in inventory databases that have objects (e.g., records) that represent physical objects of interest. This section briefly presents a systematic statement for this common practice.

The **Proxy** pattern is very simple [76]: the **Proxy** is an object that represents another object (the **Subject**). "Representation" means that the **Proxy** presents the same interface as the **Subject**, and therefore clients can use the **Proxy** "as if" it were the actual **Subject**. (Figure 8) The **Proxy** may provide services beyond those provided by the **Subject**, such as communication services (e.g., across a network, the **RemoteProxy** pattern), or access control, caching, or filtering.

In the case where the **Subject** is a software entity, the **Proxy** interacts with the **Subject** on behalf of the client, calling methods on the **Subject**. This is implemented by static or dynamic linking, or by runtime protocols such as provided by an ORB.

The goal is to extend this pattern to objects that are not purely software, particularly "dumb" (or at least "silent") objects; objects that have no computing capability, such as people, rooms, and books. These objects should have proxies that follow the same fundamental design pattern as software entities, in order to exploit powerful software mechanisms, such as object oriented languages and ORBs. However, the implementations must be quite different from proxies for software entities.

Let's define a new pattern called "**ProxyForPhysicalObject**", which is a specialization of the **Proxy** pattern. (Figure 9) The **Subject** is a "**PhysicalObject**", which means that it cannot or does not have a direct connection to the distributed system. Therefore, the **ProxyForPhysicalObject** is not implementing another software interface; it is actually

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

presenting its own interface to a model (implicit or explicit) of the **Subject**. For example, a **Proxy** for a light switch presents a software interface to a software model of a physical switch with two or more states.

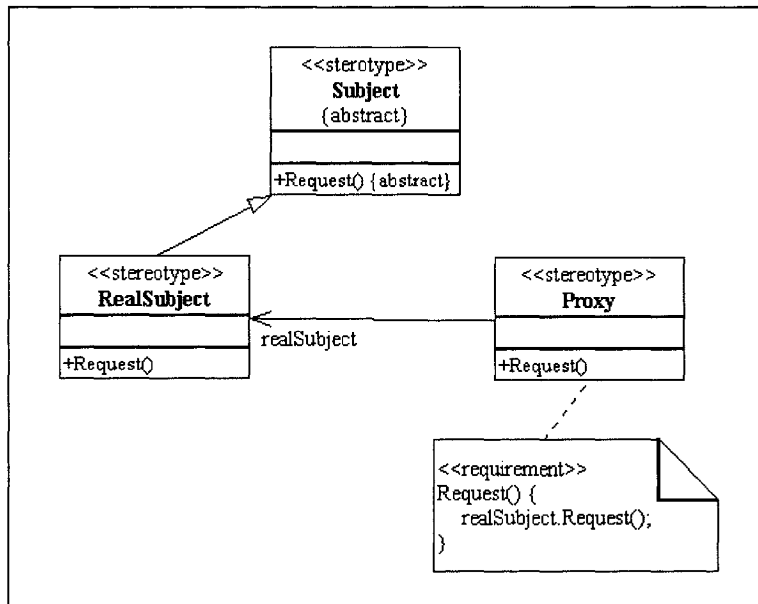


Figure 8. The Proxy pattern (diagram adapted from [76]).

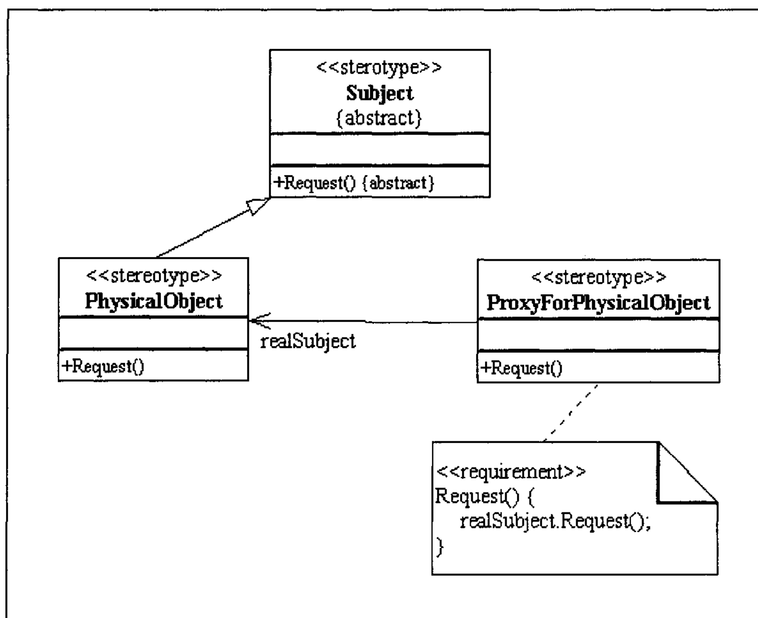


Figure 9. ProxyForPhysicalObject Pattern.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

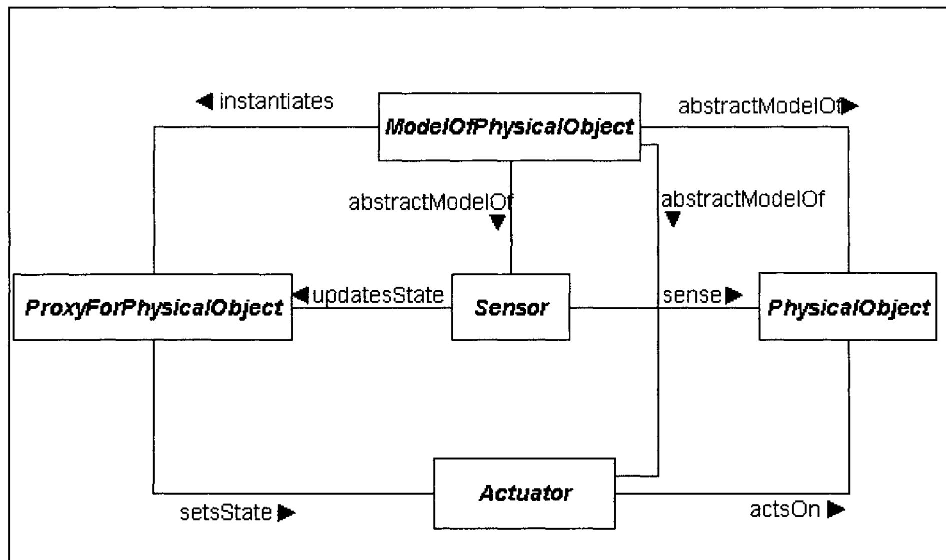


Figure 10. Associations between the ProxyForPhysicalObject and the PhysicalObject.

The **ProxyForPhysicalObject** implements accessor and other operations for the state of the **Subject**. The accessor operations are implemented by queries to the state of a software model, environmental sensors, or a combination of both. Operations that manipulate the state are implemented by updates to the model, environmental actuators, or a combination of both. (Figure 10) The **ProxyForPhysicalObject** object hides this complexity, and provides a software entity for consumers.

Clearly, the **ProxyForPhysicalObject** has additional levels of indirection compared to the simplest cases of **Proxy**, and correspondingly greater complexity. The rest of the thesis presents a model and mechanisms for managing metadata, to implement a layer of indirection.

5. Encoding Ontologies: A Formal Language for Metadata

Once the concepts of a model are identified, they must be encoded in a formal language, suitable for machine processing. Encoding in a formal language allows concepts and relationships to be stated unambiguously, and also enables proof of consistency and other properties. These proofs not only assure the validity of a single ontology, they enable proofs about multiple ontologies, e.g., to prove two ontologies are logically consistent with each other. This makes it possible to share ontologies in a decentralized system.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Many possible languages might be used for this purpose. In general, concepts and relations of an ontology are represented as statements in a subset of First Order Logic. Of course, any language as expressive as First Order Logic is undecidable, so research in Knowledge Representation has led to a variety of logics which trade off expressiveness against complexity (e.g., [18, 21, 85]).

Description Logics have been shown to provide a unifying formalism for class-based knowledge representation. Description Logics are descendants of Semantic Networks [191] and related to *frame theory* [164]. Description Logics represent knowledge about classes and logical relations, including necessary and sufficient conditions for an object to belong to a class. As a consequence, a Description Logic can automatically classify objects and discover (implicit) subsumption (inheritance) of classes. Description Logics typically work with a Knowledge Base (KB) that may contain nondeterminism and/or incompleteness. Unlike the case of the relational database model, query answering "requires the same reasoning machinery as logical derivation" ([118], p.187).

The Semantic Web suite of standards defines an XML language for encoding Ontologies, called the Web Ontology Language (OWL) [249]. The OWL-DL XML language is a subset of the OWL language that is mapped to Description Logic [115]. This means that proofs using Description Logic imply properties of the XML.

An *ontology* written in the Web Ontology Language (OWL) can be interpreted into a series of assertions to a Knowledge Base [115, 249]. When the Knowledge Base is proved logically consistent, then the ontology is valid. The formal semantics means that queries to the Knowledge Base reflect facts about the ontology. Unlike standard XML, an OWL XML document is effectively a logic program.

This technology provides a formal language for encoding ontologies, along with a universal XML format for sharing. Together, this is the necessary foundation for using ontologies in a Pervasive Computing Environment. The following section presents the additional infrastructure that must be built on this foundation.

This section summarizes the formal foundations of a language for encoding ontologies; to show and how metadata will be encoded. Section 5.1 reviews key work on formal languages to express metadata. This study uses a specific subset of first order logic called *Description Logic.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Section 5.2 summarizes the Description Logic that will be used, including a Knowledge Base and reasoning engine. Chapter 5 presents an implementation using Semantic Web technology.

5.1. Languages for Metadata

There is a choice of languages in which to write metadata, and the choice has implications for what can be expressed and computed using a given encoding. This section reviews some of the important formal languages.

Unstructured, semi-structured, or structured text can be used to express metadata. Text processing based on keywords or string (pattern) matching is relatively easy and efficient to implement, and is widely used, e.g., in search engines. String matching performs well in limited cases: essentially, when the vocabulary is controlled. However, in open and heterogeneous systems, string matching gives poor precision and recall, unless the concepts rather than the words (strings) can be matched [33, 68, 136, 210].

An alternative to string matching is to create an organized database with a logical structure that supports reasoning to answer questions. Many alternative logics have been studied and implemented in the context of databases, automatic reasoning, and programming languages.

One approach is to use first order logic (FOL) to describe the data and questions; and to use mechanical reasoning to answer questions. This approach is quite powerful, but FOL is usually impractical for automated reasoning (without human assistance). First order logic is not decidable, and the worst case complexity is exponential [233]. Research has sought to create algorithms and subsets of FOL that are more efficient.

The empirical finding is that the worst-case runtime efficiency of any correct-and-precise reasoning process increases with the expressiveness of the language. Therefore, research has aimed to develop practical systems that are as expressive as possible, while remaining tractable, at least in the "average case". This might be achieved by reducing expressiveness, allowing imprecise answers, or allowing some incorrect answers [81].

One of the most successful approaches has been relational database management systems (RDMS). Typically, a RDMS has a formal semantics that includes only conjunctions of positive ground atomic literals (which are realized as tables of tuples). These restrictions assure that answering standard SQL queries is at worst linear in the size of the database [69].

Many database systems include practical assumptions that include (e.g., see [69], p. 158):

- Closed World Assumption – facts not known to be true are assumed to be false

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- Unique Name Assumption – individuals with different names are different
- Domain Closure Assumption – there are no other individuals than those in the DB

Intuitively, the Closed World Assumption allows a reasoner to conclude that "*A* has at most *k* items", and the Unique Names Assumption that "*A* has at least *k* items" ([81], p. 9). These assumptions increase the effectiveness and efficiency of the DBMS, but they also require that *no facts can be derived other than facts in the database.*

The Ubiquitous Computing Environment is not a "closed world", and the information in the database is likely to be uncertain and incomplete. For this reason, a standard DBMS is not well suited for the metadata of such a dynamic and heterogeneous system.

Another successful approach is the Prolog logic programming language. Prolog implements reasoning with Horn clauses, a subset of propositional logic limited to conjunctions of disjunctions, where each disjunctions can include at most one positive literal ([81], p. 9). This restriction is motivated by the existence of a linear time algorithm for answering questions from a Horn clause database. However, Horn clause logic has limited expressiveness, and for this reason the Prolog programming language is augmented with "extra" (pragmatic) concepts including sets (such as, "setOf") and imperative instructions (such as, "cut" to control backtracking).

A variety of systems have implemented "Deductive Databases", which loosen some of the restrictions of the relational database model, to better deal with known false and unknown facts [22, 69, 140, 195]. These databases use various logic models, and some can derive some facts not asserted to the database. Datalog is a restriction of Prolog designed to implement logical reasoning for relational databases [22, 194, 195].

The Prolog and Datalog languages discussed above are generally considered inadequate for general purpose knowledge representation and metadata. First, they have limited ability to deal with incomplete or uncertain knowledge. Also, these systems are based on monotonic logic, i.e., the assumption that the addition of knowledge never requires the retraction of previous knowledge [85].

Research in Artificial Intelligence has investigated general purpose Knowledge Representation languages. Alternative approaches have developed, such as, *Semantic Nets* [191] and *Frame-based Systems* [164]. These formal logics seek to produce languages that are less expressive (e.g., compared to first order logic) but sufficiently general and more efficient. A

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

general purpose Knowledge Representation language can be used to express ontologies as a special case.

As discussed earlier, with many possible languages for Knowledge Representation, and it is difficult to interoperate multiple systems. This challenge has been addressed by the Knowledge Interchange Format (KIF). KIF is an ANSI standard language for exchanging knowledge representations [71]. KIF is the most widely used standard, and the Open Knowledge Base Connectivity (OKBC) standard defines an open interface for exchanging KIF [26].

KIF has a semantics that include first-order and second-order logic, and functions. Therefore, KIF is very expressive, but computational intractable in the worst case. Ontolingua is a specialized profile of KIF designed for expressing ontologies [88].

Frame Logic (F-Logic) is a language that integrates frame-based languages and first order predicate calculus [128]. F-Logic has a model-theoretic semantics and a sound and complete proof theory. F-Logic is very expressive, and can be used to represent and reason about ontologies (e.g., [44]).

In recent years, "Description Logics" (DL) have been developed to formalize and extend logic languages for knowledge representation [81]. Research on DLs has aimed to produce a language as expressive as possible while keeping the runtime polynomial, i.e., add features to get as close as possible without "falling off the complexity cliff" [81]. Figure 11 suggests the relationships between some of these logics.

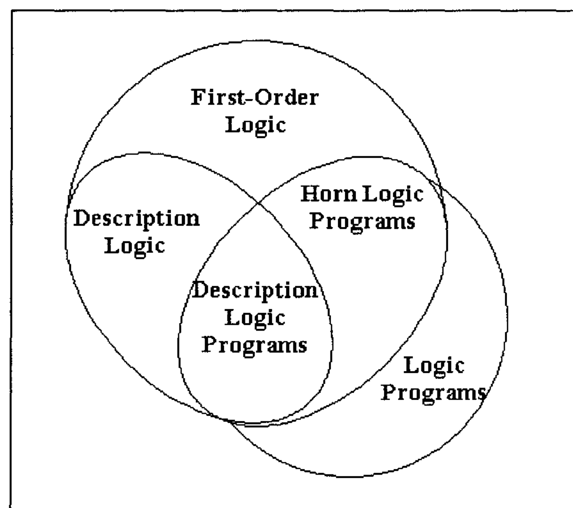


Figure 11. Sketch of relationship of some logics (adapted from [85]).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Description Logic (DL) has several advantages, including well-understood algorithms, efficient implementations of automated reasoning, and standard mappings to the Semantic Web XML languages. However, DL lacks expressive power for many important concepts. Description Logic can be extended with additional first order logic constructs that increase its expressiveness without necessarily falling off the complexity cliff (e.g., see [85]). These extensions are not considered in this study.

5.2. Description Logic

Description Logics (DL) have been shown to provide a unifying formalism for class-based knowledge representation. DLs have a tight correspondence to propositional dynamic logics (PDLs), which is a version of modal logic [211]. It has been shown that objects in DL correspond to states in PDLs, and links between objects correspond to state transitions. This correspondence has led to the application of research results from modal logic to DLs, including the development of reasoning techniques [20].

Description Logics represent knowledge about classes and logical relations, including necessary and sufficient conditions for an object to belong to a class. As a consequence, DL can potentially automatically classify objects, to discover (implicit) subsumption (inheritance) of classes. DLs typically work with a Knowledge Base (KB) that may contain nondeterminism and/or incompleteness. Unlike the case of the relational database model, query answering "requires the same reasoning machinery as logical derivation" ([118], p.187).

Description Logics are descendants of Semantic Networks [191] and related to *frame theory* [164]. Description Logics are also related to object-oriented languages: the classes and types (but not behaviors) of an object-oriented language can be stated in a Description Logic as hierarchies of concepts and roles. When a class hierarchy is expressed in a Description Logic, the model is proved *satisfiable* if and only if the class hierarchy is correct (i.e., type checking is correct). (Of course, it is not necessary to implement a general-purpose logical system to implement type checking.)

The following sections give a brief introduction to the Description Logic used in this study, along with a sketch of the implementation used. Some limitations of Description Logic are presented, and alternatives noted.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.2.1. Formal Introduction to Description Logics

Description Logics are a general class of logic are specifically designed to model vocabularies (hence the name) [20, 45, 63, 66, 67, 80, 90, 100, 111, 116-118, 183, 204]. Description Logic defines a formal language for *concepts* and relations (termed *roles*), along with a proof theory.

A Description Logic is a language for expressing factual assertions, intensional knowledge (i.e., statements about what *is* true), and queries, including satisfiability and subsumption. The elements of the language are:

- A *concept,* which represents a class, category, or entity.
- A *role* is a binary relation, which represents a property or relation among concepts
- Constructors for concept expressions, including conjunction, disjunction, and definition of relations

Systems built using Description Logic are used to create a *Knowledge Base.* A Knowledge Base (KB) is a pair, (Tbox, Abox):

- Tbox (the *terminological knowledge*): intensional knowledge; a schema defining classes, properties, and relations among classes
- Abox (the *assertional knowledge*): extensional knowledge; a (partial) instantiation of the schema, containing assertions about individuals.

Basically, the *Tbox* is the model of what *can* be true, the *Abox* is the model of what *currently is* true.

A Description Logic has a formal semantics, which can be used to automatically reason about the KB. The reasoning includes the ability to deduce answers to important questions including:

- Concept satisfiability – can concept C exist
- Subsumption – is concept C a case of concept D
- Consistency – is the entire KB satisfiable
- Instance Checking – is an assertion satisfied.

Briefly, the meaning of Description Logic can be defined with model theoretical semantics (e.g., see [20, 63, 80, 100, 102, 204]). Description Logic is defined by an *interpretation*, $I = (\Delta^I, .^I)$. Δ^I is the domain of discourse, a non empty set. $.^I$ is an interpretation function which maps:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- Every concept to a subset of Δ^I. For concept C, the interpretation maps to the set C^I.
- Every role to a subset of $(\Delta^I \times \Delta^I)$.

Concepts can be constructed from expressions. The constructors include the basic set operations (set complement, union, and intersection) and quantified role restrictions. For example, if A and B are concepts, then $(A \cap B)$ is a concept, $\neg B$ is a concept, and so on. Table 2 gives the constructors and their semantics for the family of Description Logics considered in this chapter.

Assertions (also know as "axioms") define subsumption relations between concepts ($A \subseteq B$) (i.e., $A \Rightarrow B$) and roles ($R \subseteq S$). The former define a class hierarchy, the latter defines a hierarchy for properties, e.g., to express specializations of a type of relationship. Note that the axioms can be defined for expressions, e.g., assert: $((B \wedge C \wedge \neg D) \subseteq A)$.

Simple examples of concepts and concept expressions would be:

1. Professor
2. TA
3. Student
4. (TA $\cup$ Professor)
5. Course

Simple examples of roles would be:

1. teachesClass
2. enrolledIn

Examples of concept expressions involving restrictions on roles could be:

1. Professor $\cap$ $\exists$teachesClass.Course

   (i.e., $\{ A \mid \text{Professor}(A) \wedge (\exists B \mid \text{teachesClass}(A,B) \wedge \text{Course}(B)) \}$)
2. Student $\cap$ $\exists$enrolledIn.Course
3. FullTimeStudent: (Student $\cap$ $\geq$4 enrolledIn.Course)

The first statement states that a Professor does teach a Course, the second states that a Student enrolls in a Course. The third defines a full time student as a Student who is enrolled in at least four Courses.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 2. Constructor Semantics for a simple Description Logic (see also [63, 80, 100, 102, 204]). (Concepts A, C, D; Roles R, S)

	Term	**Syntax**	**Semantics**
Concepts	Concept	A	$A^I \subseteq \Delta^I$
	Top	T	Δ^I
	Negation(C)	$\neg$ C	Δ^I - C^I
	Bottom	$\bot$	$\emptyset$
	conjunction	$C \cap D$	$C^I \cap D^I$
	disjunction	$C \cup D$	$C^I \cup D^I$
	Existential restriction	$\exists$ R.C	$\{x \in \Delta^I \mid \exists y.(x,y) \in R^I \wedge y \in C^I\}$
	Value restriction	$\forall$R.C	$\{x \in \Delta^I \mid \forall y.(x,y) \in R^I \Rightarrow y \in C^I\}$
	Quantified number restrictions	$\geq$n R.C ($\leq$, =)	$\{x \in \Delta^I \mid \#\{y.(x,y) \in R^I \wedge y \in C^I\} \geq n\}$ ($\leq$, =)
Roles	Role name	R	$R^I \subseteq \Delta^I X \Delta^I$
Axioms	Concept subsumption	$B \subseteq A$	$B^I \subseteq A^I$
	Role Hierarchy	$R \subseteq S$	$R^I \subseteq S^I$
	Role inverse	R^-	$\{ (y,x) \mid (x,y) \in R^I \}$

In Description Logics, any query is reduced to determining KB *satisfiability* (i.e., logical consistency). Thus, to answer query *Q*, one must try to prove "*not Q*" is *unsatisfiable* (i.e., the current KB cannot imply *not Q*), in which case, *Q* must be true. Clearly, this method can be very much less efficient than simpler models such as SQL. The advantage of using this general-purpose automated reasoning is that it can be used even when the KB contains non-determinism and/or incompleteness. ([118], p.187) . Table 3 gives the formal definition of satisfiability and other proofs.

Table 3. Definition of logical queries for a description logic (e.g., see also [63, 100, 204]). Concepts C, D, instance a.

Query	**For a KB, Σ = < Tbox, Abox >**
Concept satisfiability	$\Sigma \not\models C \equiv \bot$
Concept subsumption	$\Sigma \models C \subseteq D$ (i.e., $\Sigma \not\models C \cap \neg D \equiv \bot$)
Consistency	$\Sigma \not\models \bot$
Instance satisfiability	$\Sigma \models C(a)$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The automated reasoning can answer questions about the either the Tbox (the schema) or the Abox (instances). Queries about the Tbox discover classification and concept relations, queries about the Abox discover the current state of the known facts. (This study does not use reasoning about instances, but the implementation works for instances as well as concepts.)

Concept satisfiability is a proof that a concept or concept expression is logically consistent with the Knowledge Base. For example, if the KB contains the facts:

1. (Course $\cap$ $\forall$hasTA.GraduateStudent)
2. GraduateStudent $\subseteq$ (Student $\cap$ $\neg$UnderGraduate)

The first statement states that all TAs for a Course are Graduate Students. The second defines a Graduate student to be a Student who is not an undergraduate.

In this small KB, the concept *(Course $\cap$ $\exists$hasTA.UnderGraduate)* (a Course that has a TA who is an Undergraduate) is not satisfiable, because there is a contradiction with the two assertions above. A Knowledge Base is satisfiable if every concept is satisfiable.

The third logical test is subsumption. Concept *A subsumes B* if *B implies A*, according to the Knowledge Base. Axioms define some subsumption relations for concepts and concept expressions. In addition, the automated reasoning deduces subsumption implied by facts in the KB.

For example, suppose a KB has the following concepts:

1. Student
2. Graduate $\subseteq$ Student
3. UnderGraduate $\subseteq$ (Student $\cap$ $\neg$Graduate)
4. Sophomore $\subseteq$ (UnderGraduate $\cap$ $\geq$8 completed.Course $\cap$ $\leq$16 completed.Course)

In this simple example, several subsumption relations are defined. Graduate $\subseteq$ Student (i.e., Graduate is a subclass of Student), UnderGraduate is a Student but not a Graduate, and a Sophomore is an UnderGraduate who has completed at least 8 and not more than 16 courses.

Furthermore, the automated reasoning can determine additional subsumption relationships, including:

1. Sophomore $\subseteq$ Student
2. $\neg$(Sophomore $\subseteq$ Graduate)
3. Sophomore $\subseteq$ (Student $\cap$ $\leq$16 completed.Course)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

These concepts are, a Sophomore is a Student, a Sophomore is not a Graduate, and a Sophomore is an underclassman.

5.2.2. Implementation of Description Logic

Description Logics have been demonstrated to provide substantial expressive and reasoning power with real and effective implementations. Description Logics are a family of related languages that have been given conventional names. Table 4 shows the elements of the Description Logics that have been used to implement the Semantic Web.

This study uses software that implements the *SHIQ(D)* logic [10, 107]. *SHIQ(D)* logic is a specific Description Logic which is quite expressive but can be implement efficiently. The *SHIQ(D)* logic includes the constructs in all the rows except "nominals" and "value restrictions" (in Table 4, the rows labeled "S", "H", "I", "Q", and "D", but not "O" and "N"). This language includes datatypes (such as boolean, integer, etc.), which are handled as if they are predefined concepts.

Efficient algorithms for *SHIQ* are given in [117], and implemented in the FaCT (Fast Classification of Terminology) server [10, 108]. Similar algorithms are proven for *SHOQ* [116, 183], and implementations should be forthcoming. (The *SHOQ* logic is similar to *SHIQ*, but includes the constructs of "O" instead of "I" from Table 4 [116, 183]).

Briefly, these algorithms convert the assertions to a normal form which is a graph in which nodes represent a concept and edges represent relations [112]. *Satisfiability* is proved by initializing the tree with a node labeled with the concept. The node is expanded following rules that either extend the concept or add successor nodes. For a disjunction, $(C \cup D)$, the node is non-deterministically expanded to either C or D. For existential relations, a successor node is added, for universal relations the successor nodes are expanded. These steps are repeated until no more substitutions can be made or a clash is discovered. The Knowledge Base is consistent if all concepts are satisfiable.

Briefly, these algorithms convert the assertions to a normal form which is a graph in which nodes represent a concept and edges represent relations [112]. *Satisfiability* is proved by initializing the tree with a node labeled with the concept. The node is expanded following rules that either extend the concept or add successor nodes. For a disjunction, $(C \cup D)$, the node is non-deterministically expanded to either C or D. For existential relations, a successor node is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

added, for universal relations the successor nodes are expanded. These steps are repeated until no more substitutions can be made or a clash is discovered. The Knowledge Base is consistent if all concepts are satisfiable.

Table 4. Elements of Description Logic (see also [63, 80, 100, 102, 204]). (Concepts A, C, D; Roles R, S; type T, U; instance o, p, d)

DL Expressiveness	DL Syntax
$\mathcal{ALC}$, also called $\mathcal{S}$ when transitively closed primitive roles are included	A
	$\top$
	$\bot$
	$(C \subseteq D)$
	$(C \equiv D)$
	R
	R
	$(C \cap D)$
	$(C \cup D)$
	$\neg C$
	$\forall R.C$
	$\exists R.C$
$\mathcal{N}$ *(number restrictions)*	$\leq nR.\top$
	$\geq nR.\top$
	$= nR.\top$
$\mathcal{Q}$ *(quantifiers)*	$\leq nR.C$
	$\geq nR.C$
	$= nR.C$
$\mathcal{I}$ *(role inverse)*	R^{-}
$\mathcal{H}$ *(role hierarchy)*	$(R \subseteq S)$
	$(R \equiv S)$
$\mathcal{O}$ *(sets of instances)*	$\{o, p, \ldots\}$
	$\exists T.\{o, p, \ldots\}$
$(\mathcal{D})$ *(datatypes)*	U
	T
	$\exists T.d$
	$\forall T.d$

Subsumption is proved by transforming subsumption to satisfiability: concept C subsumes concept D if and only if $(D \cap \neg C)$ is not satisfiable. (See [109, 112] for a more

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

detailed explanation.) This algorithm can optimized to provide good performance for typical cases [112, 116, 117, 183].

Several implementations of Description Logic are available. The CORBA FaCT Server implements a Knowledge Base and automated reasoning for *SHIQ(D)* Description Logic [10, 107] as described here. In addition, the Java Oil package implements the mapping of DAML+OIL XML to Description Logic, i.e., it provides classes to parse DAML+OIL (and OWL) XML and asserts the correct statements of Description Logic to the FaCT Server [8, 178]. This server and interface are used as the foundation of the prototype developed in subsequent chapters.

The Racer server and Knowledge Base implements a similar set of services [94, 95]. Racer implements $ALCNH_{R+}$ Description Logic, which is very similar to *SHIQ*. Racer also provides a server [96] and the Java Oil Package implements a mapping from DAML+OIL to the Racer server. The prototype can use Racer as an alternative backend.

5.2.3. Limitations of Description Logic

While Description Logic (DL) has several advantages, including well-understood algorithms, efficient implementations of automated reasoning, and standard mappings to XML languages, DL trades-off power for efficiency. There are some concepts that cannot be expressed, or are very inefficiently expressed, using DL.

Description Logic is a language for stating classifications, i.e., *isA* and *hasA*. Essentially, DL can be used to state the "vocabulary" of the system, but cannot express other important concepts.

Description Logic does not express most quantitative concepts or numerical relations, such as "a + b" or "0 < a < 10". Quantitative concepts are essential to model many aspects of Ubiquitous Computing Environment, including spatial and temporal reasoning (e.g., [137]), resource management (e.g., [258]), and processing sensor data (e.g., [99, 131, 138]).

Description Logic has limited capability to support rules or other higher order logical statements [3, 86, 125, 216]. "If-then-else" and other rules are needed for describing many aspects of Ubiquitous Computing, including policies (e.g., [124, 258]), process and service behavior (e.g., [184]), and metarules such as constraints [77] and dependencies (e.g., [132]).

Description Logic has a very rudimentary model of queries. The basic queries of the logic need to be mapped to useful queries and results. In addition, the query and response model needs

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

to be augmented with a model of explanation, i.e., methods to determine why an item is or is not included in a result (e.g., see McGuiness for a detailed discussion of this issue [153, 157]).

In summary, Description Logic is an incomplete solution for modeling the concepts of Ubiquitous Computing. In the following chapters, higher level services are constructed on top of the FaCT server and Description Logic. These results show that Description Logic is a useful foundation for the metadata and semantic infrastructure. Nevertheless, additional languages and logic beyond DL will be needed.

5.2.4. Alternatives to Description Logic

Description Logic is not the only choice to implement the logic and reasoning needed to manage metadata. For example, Protege-2000 [170, 171], CLASSIC [156], and OntoMerge [180] have been used to implement a Knowledge Base that can load and verify DAML+OIL. Other commercial and academic systems implement KB using OWL, including Pellet [214] (which uses a different Description Logic than used here), Java Theorem Prover (JTP) [60] (which uses forward and backward chaining), Ontoprise [181] (which uses F-Logic [128]), F-OWL [237] (which uses F-Logic).

In addition, Description Logic can be extended with additional first order logic constructs that increase its expressiveness without necessarily falling off the complexity cliff. For example, "Description Logic Programming" adds certain features of Prolog and SQL, to enhance the expressiveness of DL [85]. The RuleML effort is exploring the addition of several classes of rules to DL based languages [216].

The DAML+OIL XML and OWL language is designed to be a lingua franca, that can be used with any of these systems. When fully deployed, the standard OWL XML language will be able to be used as a common format to load, update and query Knowledge Bases implemented with different logic engines.

6. How Description Logic is Used to Represent and Manage Ontologies

Description Logic can be used to manage ontologies by mapping the concepts and relations of the ontology to statements of logic. The proofs imply facts about the ontology. This section shows how Description Logic is used to represent ontologies, to prove the ontology is logically consistent, and to answer queries about the ontology.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Description Logic is used to create a Knowledge Base (KB), prove the logical consistency of the KB, and to answer queries about the KB. To prove that an ontology is valid, the concepts of the are encoded in Description Logic and asserted to a Knowledge Base. Proof that the KB is satisfiable (i.e., there are no logical contradictions) implies that the ontology is valid.

Description Logic and a Knowledge Base can also be used to answer queries. Queries are answered by encoding the query in description logic, and proving the concept is satisfiable. If the query is satisfiable, the concept described in the query is logically consistent with the ontology. Also, the Description Logic subsumption can be used to discover class-subclass relations implied by the ontology.

6.1. Representing Ontologies in Description Logic

Section 2 introduced the fundamental approach: ontologies are mapped to Description Logic through a series of translations, which allows implications from logic to be applied to the ontology. Figure 12 summarizes how ontologies are mapped to Description Logic.

An ontology (2) is encoded in a formal language (3), i.e., a language with a mapping to formal logic. The encoded ontology can be parsed to create data structures (4). The data structures are interpreted to generate statements in logic (5). Proofs about the logic implies facts about the ontology and the original model (6).

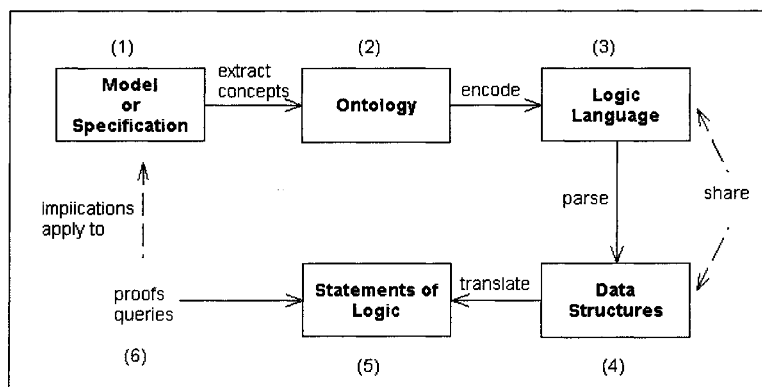


Figure 12. Overview of the use of Description Logic (Same as Figure 5 above).

To illustrate this process, consider the three classes suggested by the pseudocode in Figure 13. Class **Printer** has a variable that is a pointer to an instance of **Form**. Class

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ColorPrinter is a subclass of **Printer**. This scheme can be represented as an ontology, with three concepts, **Printer**, **Form**, and **ColorPrinter**, and a relation *supportedForm*. These concepts and relations can be encoded in many different languages.

From a given encoding of the ontology, the concepts can be represented by a data structure such as a labeled graph. Figure 14 shows an example of such a graph. Each concept (**Printer**, **Form**, **ColorPrinter**) is represented by a node, and the relationships (*subclassOf*, *supportedForm*) are represented by arcs. The details of this translation depend on the specific language used to encode the ontology, different languages might define alternative data structures.

```
Class Printer
{
   Form * supportedForm;
}
Class Form{}
Class ColorPrinter extends Printer {
  // inherits Form *supportedForm
}
```

Figure 13. Pseudo code defining three classes, e.g., as might be written in C++.

From a labeled graph such as shown in Figure 14 or a similar data structure, the ontology can interpreted to generate a set of statements of Description Logic. These statements are defined by mapping the formal semantics of the ontology language to Description Logic. Each statement is asserted to a Knowledge Base (KB), when all the statements are added, the KB is a logical representation of the ontology.

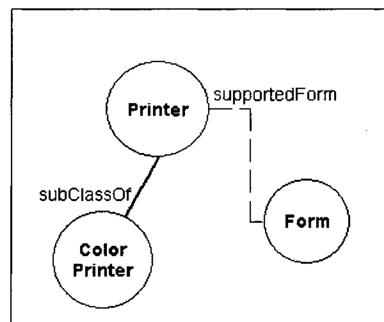


Figure 14. A graph to represent the concepts in Figure 13.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

For example, the graph in Figure 14 could be interpreted to generate five assertions in Description Logic, such as shown in Figure 15. These five statements are a *Knowledge Base (KB)*, which represents the ontology. Description Logic can be used to prove facts about this KB. For example, a proof that the KB in Figure 15 contains no contradiction implies that the definitions in Figure 13 are logically consistent.

1. DefineConcept: Printer
2. DefineConcept: Form
3. Assert: (Printer $\cap$ $\forall$supportedForm.Form)
4. DefineConcept: ColorPrinter
5. Assert: (Printer $\subseteq$ ColorPrinter)

Figure 15. Example of Description Logic corresponding to the graph in Figure 14.

This translation is the fundamental method for using Description Logic to represent and reason about ontologies. This process is deterministic: a given ontology will generate the same graph and the same statements of logic, and therefore will generate the same Knowledge Base.

6.2. Using Description Logic to Implement Proofs About Ontologies

When an ontology is represented in a Knowledge Base (KB) as described above, proofs about the Knowledge Base imply conclusions about the ontology. Section 4 presented the formal proofs defined by Description Logic, which can be applied to ontologies.

The proof of *concept satisfiability* is a proof that a concept is logically possible according to the facts of the Knowledge Base. A concept is not satisfiable if the definition of the concept implies a contradiction. When a concept is satisfiable according to Description Logic, the definition of the concept in the ontology is *valid*, i.e., it is *logically consistent with all the definitions in the ontology*. For example, in the ontology above, a proof that *satisfiable(Printer)* tests whether that the class **Printer** is logically consistent with all the definitions in the ontology.

A proof of *concept subsumption* is a proof that one concept implies another, according to the KB. When a *subsumes(A, B)* according to the KB (i.e., B implies A), then B is a subclass of A in the ontology. The Description Logic can discover subsumption relationships implied but not stated in the ontology, e.g., when two concepts are proved to be logically equivalent, this may

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

imply implicit subclass relationships (subsumption) that was not explicitly defined by statements in the ontology.

The entire Knowledge Base is *satisfiable* if all the concepts in the KB are satisfiable. This is implemented as a sequence of tests of *concept satisfiability*, one for each concept in the KB. In the example, the Knowledge Base in Figure 15 is satisfiable when:

satisfiable(Printer) AND satisfiable(Form) AND satisfiable(ColorPrinter).

When the KB is satisfiable, the ontology is *valid*. If the KB contains a contradiction, then the ontology contains a contradiction and is not valid.

6.3. Validation of an Ontology

The procedures outlined above lead to the following definition of how to determine if an ontology is *valid*. In summary,

1. The ontology is represented as a labeled graph.
2. The graph is traversed to generate a series of statements of Description Logic
3. Each statement is asserted to a Knowledge Base
4. The Knowledge Base is tested to prove it is satisfiable.

If the KB is satisfiable, then the ontology is *valid*.

This notion is defined formally in this section.

Definitions:

An *Ontology* is a directed graph, $O = (N, E)$, where N = set of nodes, E = set of edges.

Each concept in the ontology is represented as a node in N. A directed edge represents concept subsumption, or a binary relationship in the graph.

Valid Ontology:

Definition: An ontology is valid if and only if the all the concepts in the ontology ($n \in N$) are *satisfiable*.

Let Ontology A be represented by the labeled graph, $A = (N_a, E_a)$.

A Knowledge Base to represent Ontology A, KB_A, is constructed. The assertions to KB_A are Description Logic statements that define a concept for each $n_a \in N_a$ and a role or subsumption relation for each $e_a \in E_a$.

By definition,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$satisfiable(KB_A)$: **true** if $\forall n_a \in N_a : satisfiable(n_a) = true$

false otherwise

Therefore, according to the definition,

$satisfiable(KB_A) \Leftrightarrow valid(A)$.

Observations:

This definition requires that there be a standard encoding an ontology that will produce a correct labeled graph, and that the graph will be interpreted to create the correct statements of Description Logic. This translation is made possible by the definition of a formal semantics for the ontology language, and a mapping of the semantics to Description Logic. Chapter 6 discusses an implementation of this approach, using the DAML+OIL XML language, which has a formal semantics mapped to Description Logic.

The proofs depend on contents of the Knowledge Base, which can be considered the union of the assertions. Description Logic is *monotonic* (e.g., see [4]), which means that the same set of assertions will create the same Knowledge Base, regardless of the order of the assertions. As long as the same algorithm is used, the interpretation of the graph will always produce the same assertions to the Knowledge Base.

If the process defined above is implemented correctly, then two equivalent ontologies must both be valid or both be invalid, according to this test. For example, consider two ontologies, Ontology $A = (N_a, E_a)$, and Ontology $B = (N_b, E_b)$. Suppose that $A \equiv B$, i.e.,

$(N_a \equiv N_b) \wedge (E_a \equiv E_b)$

For ontology A, a Knowledge Base, KB_A is constructed. The assertions to KB_A are Description Logic statements that define a concept for each $n_a \in N_a$ and a role for each $e_a \in E_a$. Similarly, for ontology B, a Knowledge Base, KB_B is constructed. The assertions to KB_B are Description Logic statements that define a concept for each $n_b \in N_b$ and a role for each $e_b \in E_b$.

Since $(N_a \equiv N_b) \wedge (E_a \equiv E_b)$, $KB_B = KB_A$ except possibly for the order of the assertions.

Suppose Ontology A is not valid. By definition, this means that:

$satisfiable(KB_A)$ *is* ***false***, i.e., $\exists n_{inval} \in N_a : satisfiable(n_{inval}) =$ ***false***

Since $N_a \equiv N_b$, there must also be an unsatisfiable concept in B, $\exists n_{b\text{-}invalid} \in N_b : n_{b\text{-}invalid} = n_{inval}$. So, $satisfiable(n_{b\text{-}invalid})$ is ***false***, $satisfiable(KB_B)$ *is* ***false.***

Therefore,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$\neg satisfiable(KB_A) \Rightarrow \neg satisfiable(KB_B)$ and $satisfiable(KB_A) \Rightarrow satisfiable(KB_B)$.

By a similar argument,

$\neg satisfiable(KB_B) \Rightarrow \neg satisfiable(KB_A)$ and $satisfiable(KB_B) \Rightarrow satisfiable(KB_A)$.

Therefore,

$$A \equiv B \Rightarrow (satisfiable(KB_A) \Leftrightarrow satisfiable(KB_b)).$$

and by the definition of a valid ontology,

$$A \equiv B \Rightarrow (valid(A) \Leftrightarrow valid(B)).$$

6.4. Summary

This section presented the fundamental approach that is used to implement formal ontologies using Description Logic. This approach is a mechanism to represent an ontology in formal logic, and to prove the logical consistency and other properties of an ontology. The formal language and Knowledge Base are the necessary means, but they do not define the required operations on to manage *ontologies*, as opposed to Knowledge Bases.

7. Summary

This chapter presented the fundamental approach for using ontologies to represent metadata for a Ubiquitous Computing Environment. Section 3 presented the critical locality principle for ontologies, which allows a divide and conquer approach to developing ontologies. Section 4 presented a fundamental design pattern for modeling the entities of a Ubiquitous Computing Environment.

Section 5 introduced Description Logic, a subset of First-Order Logic that can be used to implement formal ontologies. Section 6 showed how ontologies can be represented in Description Logic, and how the proofs are interpreted. The ontology is written in formal logic, and to prove the logical consistency and other properties of an ontology.

The formal language and Knowledge Base are the necessary means, but they do not define the required operations on to manage *ontologies*, as opposed to Knowledge Bases. Chapter 4 defines two key algorithms: composition of two ontologies and queries on an ontology. These algorithms build on the data structures, Knowledge Base, and definition of validity defined in this Chapter.

The approach presented here can be implemented with different languages, data structures, and formal logics. Chapter 5 and presents a prototype implementation of ontologies

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

using the DAML+OIL XML language. Chapter 6 presents a prototype implementation of the composition and query algorithms. Chapter 7 evaluates the prototype.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 4. Algorithms for Managing Ontologies

1. Introduction

Chapter 3 presented an approach to develop semantic infrastructure for Ubiquitous Computing Environments. Specialized domains will create models to define their concepts, which will be related to higher level and related ontologies. The concepts of the model will be expressed in a formal ontology, which will be encoded in a formal language based on Description Logic.

Ontologies are characterized by conceptual locality and locality in use. A local Ubiquitous Computing Environment, such as a smart space, needs to maintain an ontology that is a working set of the larger universe of all ontologies. The current ontology will be used by improved infrastructure services, e.g., by augmented protocols for service registration and discovery.

Chapter 3 reviewed the basic methods for developing and representing ontologies in a formal logic, Description Logic. This chapter builds on this foundation to create two key algorithms needed to maintain and use ontologies in a Ubiquitous Computing Environment.

First, a method is proposed for semi-automatic composition of two ontologies. This algorithm enables a service to dynamically construct a local working set from a large set of ontologies, automatically maintaining logical consistency. This algorithm is a partial solution to a very difficult problem. The composition algorithm places a few natural restrictions on the design of the ontologies, and requires "hints" from administrators.

Second, an architecture and algorithm for semantic query is developed. The architecture defines a general model for advertisement, queries, and query resolution (matching). Previous algorithms for semantic matching are analyzed, and an improved algorithm is proposed.

Chapter 5 and presents a prototype implementation of ontologies using DAML+OIL. Chapter 5 presents an implementation of the composition and query algorithms. Chapter 6 evaluates the prototype.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2. An Algorithm for Loading and Composing Ontologies

Chapter 3 presented locality principles for ontologies, which imply that ontologies can be developed and used piece-wise. This thesis makes the critical observation that the ontology for a local Ubiquitous Computing Environment is a dynamically evolving working set from the larger universe of all ontologies.

In order to maintain this working set, the local environment needs to dynamically construct its ontology, importing the ontologies needed from network sources. This process is analogous to dynamic paging in a virtual memory system, except the overall set of ontologies must maintain logical consistency among themselves. Also, the loading and consistency checks must be as seamless and automated as possible.

This section develops a method for composing two ontologies. The idea is for each local environment to maintain a current system ontology, which is dynamically composed from individual ontologies. The system ontology is constructed by repeatedly adding ontologies to the system, along with definitions of relationships between the ontologies. This section presents an algorithm for loading and composing ontologies. At each stage, the combined ontology is proved to be valid, as defined in the previous chapter.

2.1. Introduction and Justification

The composition problem can be stated thus: given two valid ontologies, create a third valid ontology that correct represents the concepts in the input ontologies, plus additional constraints and relationships, if any. This can be done by a manual cross-walk of the two ontologies, to discover synonyms, resolve contradictions, and create the third ontology. But for a Ubiquitous Computing Environment, this process should be automated as much as possible.

For any two ontologies, there may be many ways that they can be composed without creating a contradiction. In general, almost any concept in one vocabulary *might* be related to any concept in another vocabulary. For this reason, ontologies usually cannot be composed without some constraints and hints from the designer and /or system administrators.

This section proposes an algorithm that requires small set of natural constraints on the organization of the ontologies, along with a set of simple hints from the designer or administrator. Given two ontologies that follow the rules, the algorithm automatically creates the complete composed ontology, propagating the implications of the hints. The algorithm detects

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

conflicts among the definitions, so the composed ontology is guaranteed to be valid, or else rejected if logically inconsistent.

2.2. Constraints on Ontologies

An ontology is naturally interpreted as a graph of classes. Each concept is a node, and subsumption (sub-class or equivalence) are the arcs. A class with no super class is the root of a graph; a class may have any number of super classes (multiple inheritance). A class may be its own ancestor, although this cannot necessarily be supported by the automated reasoning algorithms. Finally, a ontology may be a forest, with many rooted graphs, which may be connected or disjoint.

A ontology may use (i.e., *import*) elements from other ontologies in its definitions. When imported concepts are present, the algorithm is applied recursively to each imported ontology. For simplicity, it is assumed that the ontology has no references to external ontologies in this discussion. This can be achieved by fully expanding all the imports in an ontology before applying the algorithm. A singleton concept (unrelated to any other concept) is a degenerate graph. Without lack of generality, this section will consider an ontology that is a single graph of concepts with one root node.

The basic constraint on the ontologies is that they must be organized as a forest of rooted graphs or trees. Essentially, this prohibits graphs with loops, e.g., a node that is the parent of one of its own ancestors. Restricting ontologies to trees makes it possible to manipulate whole ontologies through operations on a small set of root nodes. However, the algorithm defined here could be applied to any ontology, but it would require more input, and be less seamless. The proposed constraints are natural, especially when an ontology is designed as an extension of a higher level ontology.

The composition algorithm is designed to merge two ontologies. The proposed algorithm imposes the following constraints on the ontologies:

1. The algorithm operates on the class graph. While attributes and constraints are checked and propagated, they are not used as input to the algorithm.
2. The class graph is assumed to be a forest.
3. Ontologies are merged by splicing a tree from one ontology as a sub-tree of a node of the other ontology.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The combined ontology is validated by the same algorithm that is used to validate the individual ontologies, as defined in Chapter 3.

2.3. "Hints" for Composition

In addition to hierarchies of related terms, there might be logical relationships between any other concepts in the ontologies. For example, two concepts in different ontologies may be synonyms, or one non-root class may be a sub class of some class in another ontology.

Also, it may sometimes be important to explicitly state that two concepts are, by definition, distinct (disjoint). This declaration may be necessary because two similar but distinct concepts may not have any distinguishing features within the definitions of the ontology. Figure 16 shows an example ontology that defines the concepts "Black Coffee", and "Tea". These concepts are both subsumed by the global concept "Hot Beverage", and have similar (perhaps identical) attributes in the ontologies, so they might not be logically distinct. It may be necessary or desirable to explicitly state that black coffee is not the same concept as tea, e.g., to prevent one be pruned as a redundant node.

These relations can be stated explicitly as assertions (called *axioms* in logic programming), which can be included as hints when composing the ontologies. Axioms are added as assertions to the KB when the composed ontology is validated.

Conceptually, axioms represent global constraints, but they are represented by statements the ontology language that are handled just like definitions of concepts and relations. The axioms are translated into labeled edges in the ontology graph, and then into corresponding statements of logic.

Axioms can define relationships between concepts within in a single ontology, or between concepts in different ontologies. In the former case, they may be included in the ontology (effectively, as additional relations). In the latter case, they might be in a separate ontology, which would import the ontologies that the axioms refer to.

Axioms provide a way to explicitly guide and even override the default graph composition algorithm. This mechanism should be used when there is specific domain knowledge that defines synonyms and trans-ontology relations. In this way, a community can publish standard rules for how two ontologies are related. For example, two standards organizations can harmonize their ontologies, to develop and store a comprehensive cross mapping of the concepts.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

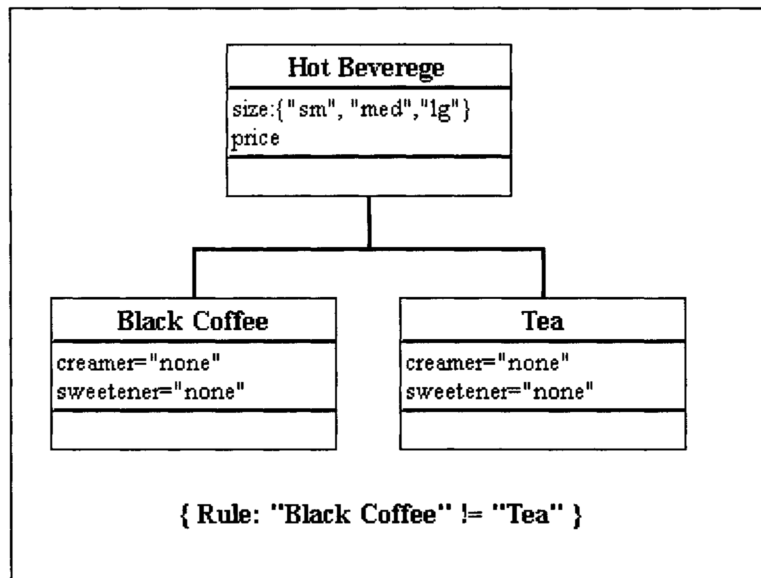


Figure 16. Example ontologies with two concepts that are similar.

2.4. The Composition Algorithm

This section presents an abstract statement of the composition algorithm using the terminology and definitions from Section 6 of Chapter 3.

2.4.1. Definitions

Ontology:

An *Ontology* is an ordered pair, $O = (N, E)$, where N = set of nodes, E = set of edges. Each term in the ontology is represented as a node in N. A directed edge represents concept subsumption or a binary relation between concepts. A *root node* has no incoming edges.

The nodes and edges of two ontologies O_1 and O_2 have unique names.

The empty ontology is the ordered pair, $O_{empty} = (\varnothing, \varnothing)$

A Valid Ontology:

A valid ontology is an ontology that has been proved consistent, e.g., using Description Logic as described in Section 6 of Chapter 3.

2.4.2. The Composition Operation: $O_1 + O_2$

Goal: Given two ontologies, create a new ontology that is the composition of the input ontologies, including relations between terms in the two ontologies.

Input:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

O_1, O_2: *Valid* ontologies. $O_1 = (N_1, E_1)$ and $O_2 = (N_2, E_2)$

links: A set, L, of ordered pairs, (n_1, n_2), where $n_1 \in N_1$ and $n_2 \in N_2$. By convention, all $n_1 \in N_1$ in L should be the root of a graph in N_1.

Output:

O_{1+2} : A *valid ontology*, with all the objects of O_1 and O_2, plus the relations stated in the *links*. If the resulting ontology contains a contradiction, then fail.

Algorithm:

The composition algorithm has three steps: merging, linking the specified root nodes, and then the composed ontology is validated to prove that no contradiction has been introduced.

Step 1: Merge O_1 and O_2 to form O_{1+2}

By definition: $O_{1+2} = O_1 \cup O_2$

$O_1 \cup O_2 = (N_1 \cup N_2, E_1 \cup E_2)$

Note: The union can always be computed, but the result may potentially contain logical contradictions. Optionally, the ontology $O_1 \cup O_2$ can be validated at this stage.

Step 2: link related hierarchies, specified by the set *links*.

For each $(n_1, n_2) \in L$, assert $subClassOf(n_1, n_2)$ to O_{1+2} i.e., create a new edge in the graph. After this step, the composed ontology is:

$O_{1+2} = (N_1 \cup N_2, E_1 \cup E_2 \cup L)$

Note: Assuming the elements of the links are valid (i.e., each $n_1 \in N_1$ and $n_2 \in N_2$) this operation can always be computed, but is possible for the links to create a contradiction. Again, the ontology $O_1 \cup O_2$ can be validated at this stage.

Step 3: Validate the composed ontology, O_{1+2}.

If O_{1+2} is valid then SUCEED

else FAIL

Done

2.4.3. Axioms: Constraints and Trans-Ontology Relationships

When two ontologies are composed, there may be additional logical constraints or relationships that apply to the combined ontologies. In the particular, it is often necessary to define relationships between concepts in the two input ontologies.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

These relationships must be added to the ontology as additional edges. This is accomplished by creating a small, fragmentary ontology that states the axioms, which is composed into O_{1+2} using the algorithm described above.

The overall composition process may involve several compositions, to construct the desired ontology.

2.5. Some Properties of the "Compose" Operation

The composition algorithm defines an operation on the universe of *valid ontologies*, as defined in Section 6 of Chapter 3. This operation exhibits some simple properties that follow from the definitions and the monotonicity of Description Logic.

2.5.1. Composition of Ontologies Is Not Closed

The composition of ontologies is not closed over the set of valid ontologies, i.e.

$valid(O_1) \wedge valid(O_2)$ does not imply $valid(O_{1+2})$.

This is true because O_{1+2} may have relationships between concepts in O_1 and O_2 which create a contradiction that is not implied by either O_1 or O_2 alone.

To illustrate how this may occur, consider the two simple ontologies in Figure 17. Ontology 1 defines five concepts, and relationships between them. **C4** is defined to be subsumed by **C2**, and **C5** is defined to be subsumed by **(C3 AND C4)** (a form of multiple inheritance). In addition, a constraint is defined on concepts **C1** and **C2**: a **C1** cannot be a **C2**. This constraint means that an object can be either a **C1** or a **C2**, but not both a **C1** and a **C2**. Ontology 1 is valid, therefore it contains no contradictions.

Ontology 2 defines a relationship between two concepts **C1** and **C3**, which are the same concepts as **C1** and **C3** in Ontology 1. In Ontology 2, **C3** is defined to be subsumed by **C1,** i.e., a **C3** is a **C1**. Ontology 2 is a valid ontology.

When these two ontologies are composed, they form a third ontology, Ontology 1+2 (Figure 18). Both the input ontologies define relations between concepts **C1** and **C3**, so Ontology 1+2 has all these relationships.

The logical analysis of Ontology 1+2 discovers a contradiction. The reasoning might be as follows. Given that **C1** and **C2** are defined to be disjoint, from the subsumption relationships, **C3** and **C4** must also be disjoint. **C5** is defined to be subsumed by **(C3 AND C4),** but it is not possible to satisfy both **disjoint(C3, C4)** and **(C3 AND C4).** Therefore, the composed ontology is not valid. Figure 19 sketches this reasoning.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

This example shows that the union of two valid ontologies is not necessarily valid. The contradiction occurs because *the input ontologies have contradictory statements about at least one concept(s)*. Note that the inconsistency was *implied*: constraints on **C1** and **C3** created a contradiction in the definition of **C5**. The logic discovered the contradiction, even though neither input ontology contained contradictory statements containing **C5**.

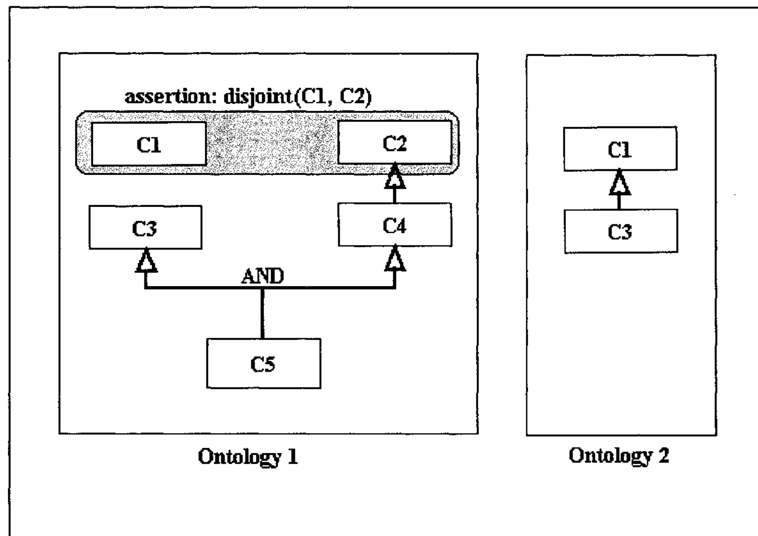


Figure 17. Two simple ontologies, each is valid.

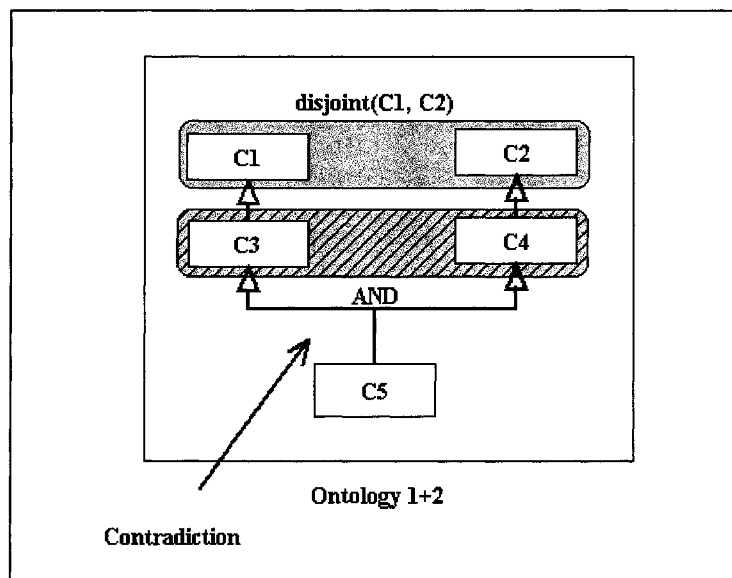


Figure 18. When composed, a contradiction is created.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

NOT (C1 AND C2)
C4 is a C2
C3 is a C1
Therefore: NOT (C3 AND C4)
C5 is a (C3 AND C4)   Contradiction

Figure 19. Deductions from the input ontology lead to a contradiction.

2.5.2. The Compose Operation is Commutative

While two composed ontologies are not always *valid*, when O_1+O_2 is valid, then O_2+O_1 must also be *valid*, i.e.,

$$valid(O_{1+2}) \Leftrightarrow valid(O_{2+1})$$

Intuitively, O_{1+2} is the same set as O_{2+1}, so they are either both valid or both invalid. Formally, this follows from the definitions of validity and the monotonicity of Description Logic.

Consider two ontologies, $O_1 = (N_1, E_1)$ and $O_2 = (N_2, E_2)$, and a set of links, L.

The composition operation, $compose(O_1, O_2, L)$ creates the ontology O_{1+2}, which is defined to be:

$$O_{1+2} = (N_1 \cup N_2, E_1 \cup E_2 \cup L)$$

Similarly, $compose(O_2, O_1, L)$ creates the ontology O_{2+1}, which is defined to be:

$$O_{2+1} = (N_2 \cup N_1, E_2 \cup E_1 \cup L)$$

These two graphs will produce the same set of statements in Description Logic, except possibly in a different order. From the fact that Description Logic is monotonic, any inference from KB_{1+2} will be valid for KB_{2+1}. Therefore,

$$satisfiable(KB_{1+2}) \Leftrightarrow satisfiable(KB_{2+1}.).$$

and by the definition in Chapter 3,

$$valid(O_{1+2}) \Leftrightarrow valid(O_{2+1})$$

2.5.3. The Compose Operation Is Associative

The composition operation also has an associative property, i.e.,

$$valid((O_{1+2})+O_3)) \Leftrightarrow valid\ (O_1+(O_{2+3})).$$

This property follows from an argument similar to above.

Consider three ontologies $O_1 = (N_1, E_1)$, $O_2 = (N_2, E_2)$, $O_3 = (N_3, E_3)$, along with links L_{1+2}, L_{2+3}, L_{1+3}.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The three ontologies are composed in two steps. To construct $O_{(1+2)+3}$,

$O_{1+2} = compose(O_1, O_2, L_{1+2}) = (N_1 \cup N_2, E_1 \cup E_2 \cup L_{1+2})$.

Assume that $valid(O_{1+2})$ is ***true***, i.e., $satisfiable(KB_{1+2})$ is **true**.

The second step is:

$O_{(1+2)+3} = compose(O_{1+2}, O_3, (L_{1+3} \cup L_{2+3})) =$

$((N_1 \cup N_2) \cup N_3, (E_1 \cup E_2 \cup L_{1+2}) \cup E_3 \cup L_{1+3} \cup L_{2+3}) =$

$((N_1 \cup N_2 \cup N_3), (E_1 \cup E_2 \cup E_3 \cup L_{1+2} \cup L_{1+3} \cup L_{2+3}))$.

Assume that $valid(O_{(1+2)+3})$ is ***true***.

To construct $O_{1+(2+3)}$, two similar compositions are done.

$O_{2+3} = compose(O_2, O_3, L_{2+3}) = (N_2 \cup N_3, E_2 \cup E_3 \cup L_{2+3}$

and,

$O_{1+(2+3)} = compose(O_1, O_{2+3}, (L_{1+3} \cup L_{1+3})) =$

$((N_2 \cup N_3) \cup N_1, (E_2 \cup E_3 \cup L_{2+3}) \cup E_1 \cup L_{1+3} \cup L_{1+2}) =$

$((N_1 \cup N_2 \cup N_3), (E_1 \cup E_2 \cup E_3 \cup L_{1+2} \cup L_{1+3} \cup L_{2+3}))$.

This is the same set as $(O_{(1+2)+3})$, except for the order. Therefore, $valid(O_{1+(2+3)})$.

From the definition of valid and the monotonicity of Description Logic,

$satisfiable(KB_{(1+2)+3}) \Leftrightarrow satisfiable(KB_{1+(2+3)})$ and so $valid(O_{(1+2)+3}) \Leftrightarrow valid(O_{1+(2+3)})$.

Note that this property implies that, for any O_{1+2+3} composed as described here,

$valid(O_{1+2+3}) \Rightarrow valid(O_{1+2}) \wedge valid(O_{1+3}) \wedge valid(O_{2+3}) \wedge valid(O_1) \wedge valid(O_2) \wedge$
$valid(O_3)$.

2.5.4. The Compose Operation Is Not Transitive

From the non-closure property, we can see that the composition operation is not transitive. That is:

$valid(O_{1+2}) \wedge valid(O_{2+3})$ does not imply $valid(O_{1+3})$.

Essentially, even if $(valid(O_1) \wedge valid(O_2) \wedge valid(O_3) \wedge valid(O_{1+2}) \wedge valid(O_{2+3}))$, it is still possible for O_{1+3} to have a contradiction as discussed in section 6.5.1 above.

Similarly, $valid(O_{1+2}) \wedge valid(O_{2+3})$ does not imply $valid(O_{1+2+3})$.

2.6. Summary of Composition

This section defined an algorithm to compose two ontologies with minimal input from humans. The key to the algorithm is a consistency check using the formal semantics of the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ontology. As the example showed, composing related ontologies can create contradictions, even when the input ontologies are perfectly valid.

This algorithm enables dynamic update of ontologies, maintaining logical consistency of the combined ontologies. This algorithm is a crucial foundation for maintaining a "current ontology" to represent the local "working set" for a Ubiquitous Computing Environment. Applications and services use the local ontology to automatically configure the local environment.

3. Design and Algorithms for Semantic Query

Section 2 developed the key algorithm that enables a local environment to maintain a working set ontology. The primary use for the ontology is to query to discover sets of concepts with specific properties. This section defines an algorithm for queries on ontologies. These queries use Description Logic to state and answer the queries.

This algorithm builds on earlier work. Section 3.1 gives background, defines of semantic matching, and reviews earlier work. Section 3.2 presents an improved algorithm that combines and extends ideas from the literature.

3.1. Definitions and Background

At the core of the semantic discovery process, it is necessary to identify a set of concepts that are "conceptually similar" to a query. This is termed *matchmaking* or *semantic matching*. This section presents an algorithm that uses the current ontology and Description Logic to implement a semantic query.

The essence of this approach is to define heuristics for converting a simple input query into some query or queries to the Knowledge Base that reflect the intention of the calling program. The heuristics define rules for judging which nodes are *logically consistent* with the query. The rules define a "similarity metric"; a region of the taxonomy graph that matches (is similar to) a given query, based on the definition of the ontology. The heuristics define a set of conditions can be implemented as a set of queries in Description Logic.

This section develops these heuristics and provides the formal statement of the match.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.1.1. Goals and Definition of Semantic Match

The general goal is to provide a service that accepts a relatively simple and general query, and uses the information encoded in the ontology and the Knowledge Base to retrieve a set of concepts that *match* the query according to some heuristic criteria.

The ontology and Knowledge Base supports a range of queries; a calling program could frame many kinds of query. A query might be for a name or pattern, similar to a relational database query. Alternatively, the caller could retrieve the whole ontology (e.g., as a graph), and perform custom algorithms.

If the query requires too much knowledge of the system configuration it will be fragile in the face of changes in the system. Also, in an open system, it is difficult to assure that all parties are using the same algorithms. For these reasons, it is not desirable to require the calling program to deal with ontologies directly.

The goal of the *semantic match* is to try to get the best of all worlds: the user issues a relatively simple query, while the service executes a complex retrieval using the graph and Knowledge Base, to return an answer that reflects the relationships in the current ontology.

In this approach, an ontology defines the taxonomy of the universe of services available. The goal is to define a heuristic to match a query concept with a set of concepts of the ontology. When the ontology is interpreted as a graph with a node for each concept, the match seeks to find a region of the graph that is "near to" the input concept, based on the logic of the ontology.

A primary goal of the semantic match is to provide a *standard* definition of similarity, which hides the details of the implementation and the current state of the ontology from the caller. This approach attempts to make the system more robust by isolating the complexity of the query from the callers. It also assures that all parties (e.g., producers and consumers) use the same heuristics, which avoids interoperation problems.

There is no accepted standard definition of a semantic match. However, there is consensus that a query for service discovery seeks to discover entities that are "consistent" with or "substitute for" the criteria in the input query (e.g., see [80, 139, 184], as discussed in the next section).

The "query" is considered as if it were a concept (or instance of a concept) of the ontology. The query can be a simple concept (e.g., a name) or the definition of a complex concept (one or more concepts with restrictions and relations). The latter case may be used to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

represent a hypothesis to be tested, i.e., to ask "is such a concept possible?" in the current system.

The query concept is checked using Description Logic using the algorithm described below. If the query is *satisfiable*, then the hypothesis is *possible.* If the query is valid, the taxonomy implied by the ontology and Description Logic can be used to discover a set of concepts that are similar to the query.

3.1.2. Review of Recent Definitions of Semantic Match

This section reviews three recent definitions of *semantic match.* These proposals offer similar definitions of semantic match, which can be implemented with ontologies and Description Logic. This thesis combines and expands these approaches.

3.1.2.1. Definition according to Gonzalez-Castillo, Trastour, and Bartolini [80]

Gonzalez-Castillo, et al. proposed a set of logical conditions that define a semantic match [80]. Gonzalez-Castillo, et al. appeal to "real-life examples like yellow pages directories, advertisement newspapers, or bulletin boards" ([80], p. 5), which they observe have descriptions with different levels of specificity. Therefore, they would define the desired match to include concepts more general than the query, more specific than the query, and neither more general nor more specific, but compatible with the query.

The set of matches is defined to be:

For any two concepts *C1* and *C2, C1* matches *C2* if:

1. *C1* is equivalent to *C2*, or
2. *C1* is a sub-concept of *C2*, or
3. *C1* is a super-concept of a concept *s*, and *s* is subsumed by *C2*, or
4. *C1* is a sub-concept of a direct super-concept of *C2* whose intersection with *C2* is satisfiable

   (adapted from [80])

These criteria can be stated in Description Logic, as discussed in section 7.2.3, below. The match criteria are summarized in Table 5.

The first two criteria are intuitively clear (and, indeed, obvious): these are the concepts that are closely related to each other in the taxonomy graph. The third criterion describes concepts that are similar because they have a child in common in the taxonomy graph. In this

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

situation, the child "inherits" from two parents, which implies that the parents may be similar in some way.

The fourth criterion defines the set of concepts that are descendants of a parent of *C2*, which are logically compatible with *C2*. The intersection of two classes is the (closed) set of all concepts that are members of both concepts, as defined by Description Logic.

Figure 20 gives a sketch of a taxonomy tree, to show which concepts that match concept *X*, according to definitions above. Concept *C2* matches according to the first rule, *C5* and *C6* match according to Rule 2, *C4* and *C1* match according to Rule 3. Concept *C3* matches by Rule 4 if there is no logical contradiction (i.e., the intersection is *satisfiable*).

The match rules may be applied to a narrower or broader area of the taxonomy graph. The match may be limited to immediate neighbors, or include more distant nodes. For example, if the search is not limited to immediate neighbors in Figure 20, then *C0* matches *X* by Rule 3 and *C7* matches by Rule 2. Gonzalez-Castillo, et al. do not state a pruning rule [80].

Table 5. Summary of four criteria for a match, a la [80].

Statement	Relation of Concepts A, B	Suggested Description Logic
(1)	A equivalentTo B	$A \equiv B$
(2)	B subclassOf A	$B \subseteq A$
(3)	S subclassOf A AND S subClassOf B	$(A \subseteq S) \wedge (B \subseteq S)$
(4)	B is a sibling or descendent of a sibling of A, and A and B are logically consistent	$parent(S, A) \wedge B \subseteq S \wedge K \models (A \cap B)$

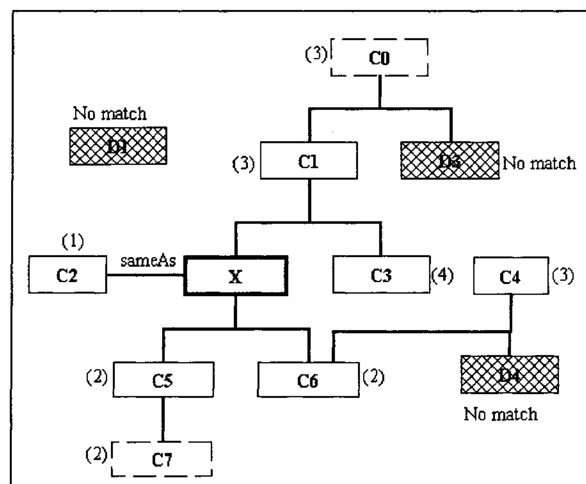


Figure 20. Sketch of the concepts that match(?, 'X'), a la [80].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.1.2.2. Definition According to Paolucci, Kawamura, Payne, and Sycara [184]

Paolucci et al. present a similar definition of semantic match [184]. In addition, they define four *degrees of match*, based on the minimal distance between the concepts in the taxonomy tree. Table 6 lists the degrees of match in order of increasing distance, i.e., *exact* match is considered a better match than *plugin*, and so on. Figure 21 illustrates the match criteria using this definition. For example, *(X subclassOf C1),* so *C1* is a match labeled *subsumes*.

They give intuitive interpretations to justify their *degrees of match*, based on ideas about service advertisement and look up. For an *advertisement, outA,* and a *request, outR*, if *outA* ≡ *outR*, then the match is obviously "exact". When *(outR subclassOf outA),* this is also an "exact" match because "by advertising outA the provider commits to provide outputs consistent with every immediate subtype of outA" ([184], pp. 339-340). If *(outR* $\subseteq$ *outA),* then the match is labeled "plugin" because "outA is a set that includes outR" ([184], p. 340). When *(outA* $\subseteq$ *outR)* the match is labeled "subsumes" because "the provider does not completely fulfill the request" ([184], p. 340).

While Paolucci et al. do not cite Gonzalez-Castillo, et al., their *degrees of match* correspond to some of the terms in Table 5, as shown in Table 10. The main difference is that Paolucci et al. do not include more distant nodes of the taxonomy as possible matches (e.g., *C3* and *C4* in Figure 21).

The degrees of match suggested by Paolucci et al. provide labels and a ranking criterion for the set of matches defined by Gonzalez-Castillo, et al.. This is a useful concept, which is extended below.

Table 6. Definition of "Degree of Match" from [184] and equivalents from Table 5.

Relation of Concepts A, B	**Suggested Description Logic**	**Degree of Match**	**Term in Table 5**
A = B	A ≡ B	"exact"	(1)
B subclassOf A	parent(A, B)	"exact"	(2)
A subsumes B	B $\subseteq$ A	"subsumes"	(2)
B subsumes A	A $\subseteq$ B	"plugin"	special case of (3)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Figure 21. Sketch of the concepts that match(?, 'X'), a la [184].

3.1.2.3. Li and Horrocks [139]

Li and Horrocks extend the *degrees of match* from Paolucci et al., adding a fourth "degree", which they term *intersection* [139]. Their degrees of match differ slightly from Paolucci et al., in that "subclassOf" is included in "A subsumes B", and therefore considered *plugin* rather than *exact.* Table 7 shows their definitions, which are similar to Table 5 above.

Table 7. Definition of Degree of Match [139] and equivalents from Table 6.

Relation of A, B (Description Logic)	Degree of Match	Term in Table 5
$A \equiv B$	"exact"	(1)
$B \subseteq A$	"plugin"	(2)
$A \subseteq B$	"subsumes"	(2)
$\neg(A \cap B) \subseteq \bot)$	"intersection"	Generalization of (4)

Although they did not cite Gonzalez-Castillo, et al., the "intersects" match corresponds to the fourth term of in Table 5. Li and Horrocks define a less restrictive version of the "intersection", considering any concepts in the taxonomy, not just nearby concepts.

Li and Horrocks provide similar intuitive interpretations of their degrees of match. For an advertisement A and a request R, when $A \equiv R$ the match is labeled *exact.* If $(R \subseteq A)$, then the match is labeled *plugin* because "we might expect that advertiser also provide more specific (sub-class) services" ([139], p. 336). When $(A \subseteq R)$ the match is labeled *subsume* "since an advertiser might also provide some more specific (super-class) services" ([139], p. 336). And

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

when $(\neg(A \cap R) \subseteq \bot)$ the match is labeled "intersect" which is a weak match because "it only says that the advertisement is not incompatible with the request" ([139], p. 336).

Figure 22 illustrates the concepts that match according to Li and Horrocks. Since there is no restriction on the concepts that might be considered of this match, *any concept* in the taxonomy potentially could match, if the intersection is satisfiable.

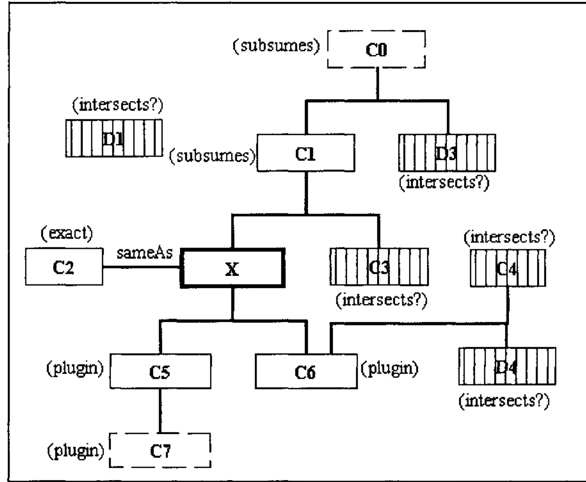


Figure 22. Sketch of the concepts that match(?, 'X'), a la [139].

3.1.3. Statement of the Semantic Matches in Formal Logic

The definitions of semantic matching discussed above were developed from intuitions about service composition. Although stated in different terms, these three definitions can be formally defined using Description Logic. These formal statements make clear similarities and differences between the definitions.

Using Description Logic, a semantic match is for a concept, *c*, can be stated as a set of concepts that meet certain logical criteria. Figure 23 - Figure 25 state the matches defined by Gonzalez-Castillo, et al. [80], Paolucci et al. [184], and Li and Horrocks [139] in Description Logic, based on the definitions summarized in Table 5 - Table 7.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

For Knowledge Base K, with concepts C,

$$
\begin{aligned}
&Match\ (c, d) = \\
&\quad \{ c, d \in C: \\
&\qquad (c \equiv d) && (1) \\
&\qquad \vee\ (c \sqsubseteq d) && (2) \\
&\qquad \vee\ (\exists s \in C:\ s \sqsubseteq c \wedge s \sqsubseteq d && (3) \\
&\qquad \vee\ (\exists\, s \in C: \text{parent}(c, s) \wedge d \sqsubseteq s \wedge \\
&\qquad\quad K \models (c \sqcap d)\,) && (4) \\
&\quad \}
\end{aligned}
$$

where $parent(c, s)$ is defined:

true if $s \neq c \ \wedge \{ \forall t \in C:\ c \sqsubseteq t \wedge t \sqsubseteq s \Rightarrow c = t \vee t = s \}$

false: otherwise

Figure 23. A formal statement of the semantic match from Gonzalez-Castillo, et al. [80] (compare to Table 5).

For Knowledge Base K, with concepts C,

$$
\begin{aligned}
&Match\ (c, d) = \\
&\quad \{ c, d \in C: \\
&\qquad (c \equiv d) && (exact) \\
&\qquad \vee\ parent(c, d) && (exact) \\
&\qquad \vee\ (c \sqsubseteq d) && (subsumes) \\
&\qquad \vee\ (d \sqsubseteq c) && \text{(plugin)} \\
&\quad \}
\end{aligned}
$$

where $parent(c, s)$ is defined:

true if $s \neq c \ \wedge \{ \forall t \in C:\ c \sqsubseteq t \wedge t \sqsubseteq s \Rightarrow c = t \vee t = s \}$

false: otherwise

Figure 24. A formal statement of the semantic match from Paolucci et al. [184] (compare to Table 6).

For Knowledge Base K, with concepts C,

$$
\begin{aligned}
&Match\ (c, d) = \\
&\quad \{ c, d \in C: \\
&\qquad (c \equiv d) && (exact) \\
&\qquad \vee\ (d \sqsubseteq c) && (plugin) \\
&\qquad \vee\ (c \sqsubseteq d) && (subsumes) \\
&\qquad \vee\ (K \models (c \sqcap d)\,) && (intersects) \\
&\quad \}
\end{aligned}
$$

Figure 25. A formal statement of the semantic match from Li and Horrocks [139] (compare to Table 7).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The definitions in Description Logic make clear similarities and differences among the proposed definitions. The definitions agree on the "obvious" equivalence and hierarchical relationships, which are expressed as subsumption relationships. The different definitions suggest several "less obvious" relations, which are defined in more complex logical expressions (e.g., (3) and (4) in Figure 23).

Figure 26 illustrates a taxonomy tree showing the concepts that match 'X' according to the three definitions. There is consensus on the "obvious" matches, with a variety of proposals for the other matches. The following section develops a definition that builds on these proposals, with an original formulation for the "non-obvious" relations.

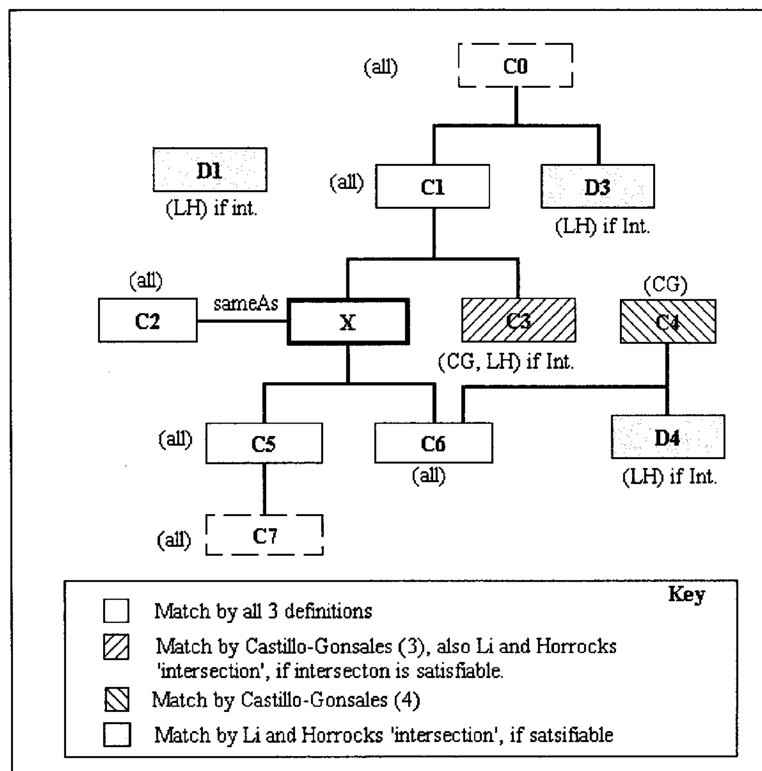


Figure 26. Illustration of the three definitions of semantic match.

3.3. Improved Definition of Semantic Match

This section presents an improved definition for semantic match. This definition builds on the work cited in the previous section, with important extensions.

Overall, the semantic match defines a similarity metric, in which the similarity of two concepts is closely related to a notion of "substitutability" of services, similar to the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

compatibility of software modules. The degrees of match are interpreted as greater or lesser degrees of substitutability. There might be other valid interpretations of the semantic match, but "degree of substitutability" is a useful aid to understanding the algorithm and results.

Table 8 gives a summary of the proposed definition of semantic match, with a label for the *degree of match* akin to the definitions cited above. In this definition, the *exact, plugin,* and *subsumes* matches are consistent with Paolucci et al. [184] as modified by Li and Horrocks [139]. These relations are intuitively clear and can be expressed in simple statements of Description Logic, as in Figure 27. Figure 28 shows the matches for this new definition, labeled with a degree of match from Table 5.

The fourth match is based on the third condition stated by Gonzalez-Castillo, et al. [80]: two concepts that have the same child. This match is given the label *coparent* and ranked as the fourth "degree of match". This condition can be stated in Description Logic, as discussed in section 3.2.1 below.

Finally, a new match is defined, which is called *substitutable.* This is an extension and refinement of the *intersection* match from Gonzalez-Castillo, et al. [80] and Li and Horrocks [139]. The definition of this match is presented in section 3.2.2. below.

Table 8. Summary of the extended definition of semantic match.

Relation of A, B (Description Logic)	Degree of Match	Term in Table 5
$A \equiv B$	"exact"	(1)
$B \subseteq A$	"plugin"	(2)
$A \subseteq B$	"subsumes"	(3)
$S \subseteq B \wedge S \subseteq A$ (See below for the complete definition.)	"coparent"	(3)
$A \subseteq S \wedge B \subseteq S \wedge$ *compatible* (A, B) (See below for the complete definition.)	"substitutable"	(4)

For Knowledge Base *K*, with concepts *C*,

Match (c, d) =

$\{ c, d \in C:$

$(c \equiv d)$ *(exact)*

$\vee\ (c \subseteq d)$ *(plugin)*

$\vee\ (d \subseteq c)$ *(subsume)*

$\}$

Figure 27. Formal statement of three simple match conditions.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

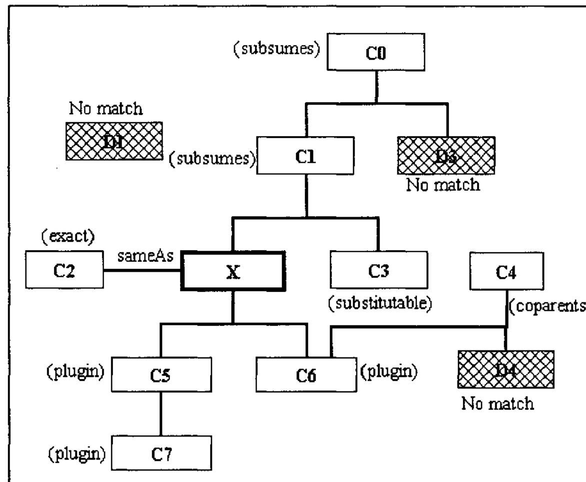
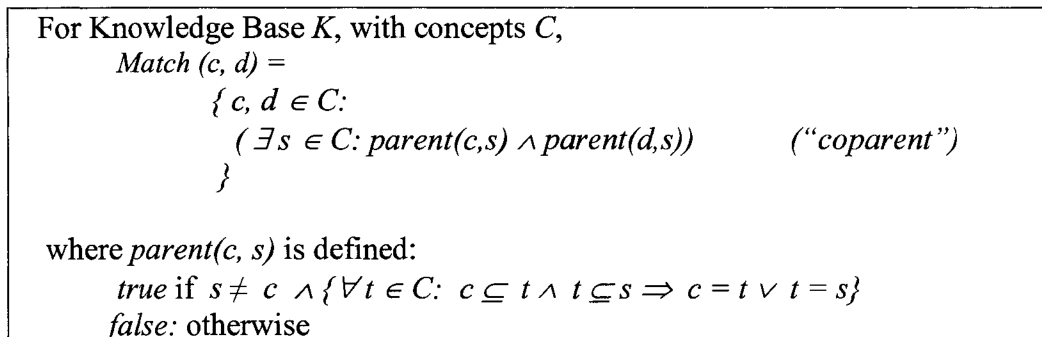


Figure 28. Sketch of the classes that match 'X'.

3.2.1. The *Coparent* Match

The *coparent* match is defined as a concept that is the parent of a child of 'X'. This can be stated in Description Logic, as shown in Figure 29.

In Figure 28, *X* matches *C4* by this relationship, because *((C6 subclassOf X) AND (C6 subclassOf C4)).* Intuitively, *X* and *C4* must be logically compatible (or at least not contradictory), or else *C6* could not be valid.



For Knowledge Base *K*, with concepts *C*,

Match (c, d) =

$\{ c, d \in C:$

$(\exists s \in C: parent(c,s) \wedge parent(d,s))$ (*"coparent"*)

$\}$

where *parent(c, s)* is defined:

true if $s \neq c \wedge \{\forall t \in C: c \subseteq t \wedge t \subseteq s \Rightarrow c = t \vee t = s\}$

false: otherwise

Figure 29. The *coparent* match.

A concept identified by this heuristic may or may not be a "reasonable" result. *C6* might be a restricted case of *X* and *C4*, such that *C4* (and *C6*) might or might not be a "good" match for *X*.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

For example, consider a simple taxonomy such as Figure 30. In this set, the concept "Prius" is a sub class of both "Automobile" and "CarriesPassengers". Using the coparent rule, *match(Automobile,CarriesPassengers)* is **true**, i.e., "things that carry passengers" might be a substitute for "automobile". This inference seems intuitively reasonable.

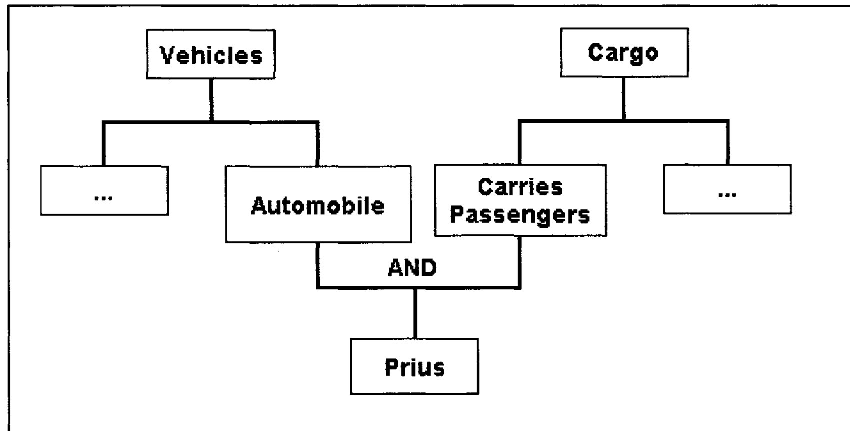


Figure 30. An ontology in which "CarriesPassengers" matches "Automobile".

However, this inference is not always so reasonable. Consider an alternative taxonomy, such as Figure 31. By analogous reasoning as above, from the fact that "Orangutang" is a sub class of both "Orange things" and "Furry things", the conclusion is that *match(Orange,Furry)* is **true**, i.e., *furry* things may substitute for *orange* things. It is questionable whether this match is reasonable for most purposes.

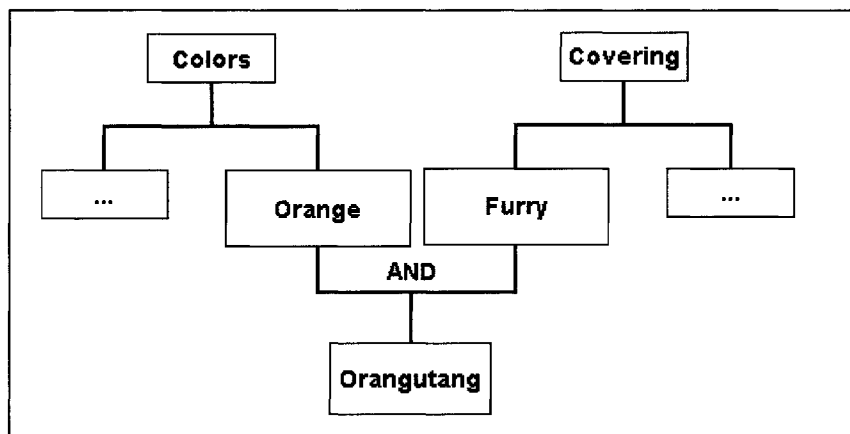


Figure 31. An analogous ontology in which "Orange" matches "Fur".

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.2.2. The *Substitutable* Match

The overall goal of this match rule is to discover concepts that are distantly related to the target concept, and to select from them ones that are compatible with the target. The definition of this match has two components: a set of concepts to consider, and a consistency criterion. The former defines a set of concepts "nearby", the latter checks whether a given candidate is logically consistent with the desired match. (The concepts found by the other rules are guaranteed to be consistent because of the subsumption relations.) These two aspects are considered in turn.

The proposals discussed in section 3.3.1 define "compatibility" based on the logical intersection, as defined by Description Logic, which has significant shortcomings. This section proposes a new definition for compatibility that addresses these problems.

3.2.2.1. The Candidate Concepts

For the *substitutable* match, the candidates are defined to be the set of concepts that are related by a common ancestor, i.e., $(d \subseteq s \wedge c \subseteq s)$. From the point of view of concept c, this is the set of siblings or cousins, descendants of s. (By definition, this set overlaps with the other sets defined above: the set includes c itself, the descendants of c, and some ancestors of c.)

The set of candidates can be limited by a pruning rule. Gonzalez-Castillo, et al. [80] propose to limit the search to descendants of a parent, i.e., for concept c, consider only its siblings and their descendants. On the other hand, Li and Horrocks [139] do not state any limit, so their rule might consider any concept in the Knowledge Base, i.e., s might be a very distant ancestor of c and/or d. If the top concept is included as the root of all concepts, then s could be any concept in the Knowledge Base in this definition.

A wider set of candidates enables less "obvious" matches to be discovered, at the cost of a larger space to search. The best pruning rule depends on the content of the Knowledge Base and the intentions of the query. Ideally, this choice should be an adjustable parameter to the query.

The most important goal of the pruning is to eliminate candidates that have only trivial relationship to each other, e.g., concepts related through a very distant ancestor in the taxonomy, which must represent only a very general and tenuous similarity. This goal can be achieved by introducing additional tests on the candidate ancestors of c and d.

A simple restriction is proposed here. The two candidates' concepts must have parent concepts that are related by subsumption, as in:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$parent(c, s_1) \wedge parent(d, s_2) \wedge s_2 \subseteq s_1$

In the test of *match(c,d)*, when *c* and *d* have a common ancestor, the parent of *c* and parent of *d* are tested for subsumption. It the parent of *c* does not logically subsume the parent of *d*, then *d* is not a match for *c*, and vice versa.

Essentially, this condition tests for the case where two concepts are "partly related". This occurs when the two concepts are defined as expressions, and one is a related, but more restricted expression than of other. (In a class hierarchy, this is a case where one or both of the classes is defined by multiple inheritance: it is important to consider all of the parents of the class.)

To illustrate this idea, consider the taxonomy shown in Figure 32, part of a concept hierarchy about devices. In this example, the concepts describe capabilities of devices. A device may have the capability to provide **HardCopyInput** or **HardCopyOutput**. The concept **HardCopyIO** is defined as the conjunction of the input and output concept, i.e., a device that provides *both* **HardCopyInput** and **HardCopyOutput**.

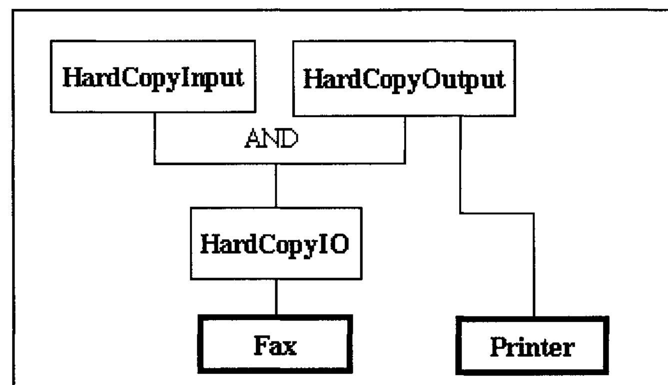


Figure 32. A simple ontology.

Intuitively, the match should reflect which concepts (in this case, devices) potentially could *substitute* for one another. In this example, a **Fax** may substitute for **Printer**, but a **Printer** cannot substitute for **Fax** (with respect to hard copy): the **Fax** can provide **HardCopyOutput**, but the **Printer** cannot provide **HardCopyInput**. Therefore, if the query *match(A,B)* means "can *A* substitute for *B?*", then *match(Fax, Printer)* should be **true**, while *match(Printer, Fax)* should be **false**.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Applying the rule stated above to the example, if *c* is **Printer,** s_1 is **HardCopyOutput**, *d* is **Fax,** $s2_1$ is **HardCopyIO**. Since $(s_2 \subseteq s_1)$ is **true**, *match(Fax, Printer)* is *true* (i.e., **Printer** substitutes for **Fax**). But $(s_1 \subseteq s_2)$ is *false*, so *match(Printer, Fax)* is **false** (i.e., **Fax** does not substitute for **Printer**).

Note that this situation is not limited to immediate parents. Figure 33 shows a taxonomy tree where the asymmetry occurs at a lower node. In this tree, *match(C,B)* and *match(B,C)* are both *true*, but *match(C,D)* is **true**, while, *match(D,C)* is **false**.

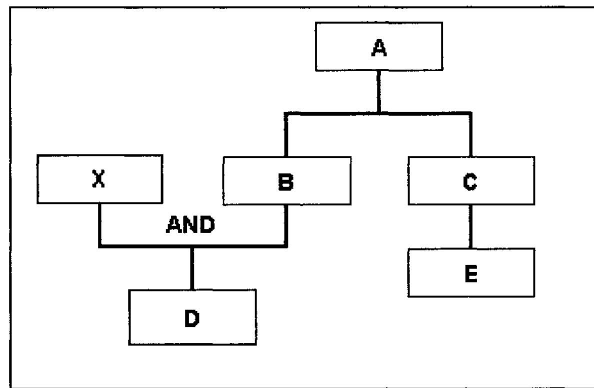


Figure 33. A simple tree in which match(D,C) is false.

3.2.2.2. The Consistency Check: Statement of the Problem

The candidates discovered by the rule defined above must be screened to determine if they are logically consistent with the desired match. The general strategy is to formulate a query to the Knowledge Base that will test the intended criteria, e.g., “substitutability”. This is done by creating a expression in Description Logic, which is tested for satisfiability. This section proposes a heuristic for constructing a logical expression based on the two concepts to be tested.

Both Gonzalez-Castillo, et al. [80] and Li and Horrocks [139] propose to use the *intersection* of the concepts, as defined by Description Logic [109, 117]. That is, they propose to test the compatibility of two concepts *A* and *B* using the Description Logic query, “*satisfiable(A* $\cap$ *B)?*” While simple to state, it turns out that this query does not reflect the intuition of “substitutable” very well.

Description Logic does not impose a “closed world” restriction on the Knowledge Base. In this context, this means that the absence of evidence can not be deduced to be evidence of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

absence. Since the satisfiability test will fail only if a logical contradiction is proved, many "incompatible" concepts can pass the intersection test. When this happens, the result of the query does not match the intuitive meaning of "substitutable."

The heart of the problem is that the intersection query is a weak check for conflicts between properties. Recall that each concept may be defined to have zero or more properties (termed "Roles" in Description Logic, e.g., see [109, 117]). These properties define variables and relations to other concepts. The relations may be quantified, and the filler (i.e., the target of the relation) may be a boolean expression (recursively including concepts with restrictions). Table 9 lists some of the properties and restrictions that may be defined for a concept.

To illustrate the challenge, consider the two concepts in Figure 35. The intersection is satisfiable because the assertion that a mother has sons does not imply that she does *not* have daughters, and vice versa. However, the intuitive conception of "substitution" seems to require a different interpretation, i.e., a "closed world" assumption that a mother of 2 boys cannot "substitute" for a mother of 3 girls.

Table 9. Class constructors for Roles in Description Logic (see also [63, 80, 100, 102, 204]). (Concept C; Role R)

Description Logic Syntax	Restriction
$\forall R.C$	Universal Role
$\exists R.C$	Existential Role
$\leq nR.C$	Cardinality
$\geq nR.C$	"
$= nR.C$	"
(datatype properties)	e.g., int, boolean, etc.

```
Concept 1:  // parent of at least 2 boys
  ≥ 2 hasKid.Son
Concept 2:  // parent of at least 3 girls
  ≥ 3 hasKid.Daughter
(Concept 1  ∧ Concept 2):
   (≥ 2 hasKid.Son  ∧  ≥ 3 hasKid.Daughter)
```

Figure 34. Description Logic is not closed.

To illustrate this in the context of a Ubiquitous Computing Environment consider the taxonomy shown in Figure 36. Using the definition of the intersection test based on Description

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Logic, the term (**Fax** $\cap$ **Printer**) is tested by testing the logical conjunction of the definitions of the two concepts. The concept (**Fax** $\wedge$ **Printer**) would be defined, and tested for satisfiability.

The concept (**Fax** $\wedge$ **Printer**) is defined by expanding the definition of the two concepts, to form a expression something like:

Define Concept: FaxAndPrinter {
((**HardCopyInput** AND **HardCopyOutput**) AND **HardCopyOutput** AND …)
}.

This expression is a conjunction of the definition of **Fax** and **Printer**. The conjunction may have additional terms if the concepts have properties or restrictions.

Suppose the concept **Printer** is defined to have several properties, such as *printDensity*, *paperSize*, and *orientation*. The concept **Fax** might have the same or other properties, such as *baudRate*. Figure 35 shows pseudocode for how these might be defined.

The conjunction of the two concepts is implemented by expanding the definitions of the two classes, including the definitions of the properties. Figure 36 shows pseudocode for the conjunction of the concepts defined in Figure 35. This conjunction would be written in Description Logic, and tested for satisfiability.

First, a property may be defined for one concept and not for the other, e.g., *baudRate* in the example. The conjunction asserts this property for the concept, e.g., *(PrinterANDFax)* $\Rightarrow$ $\exists$*baudrate.64KBS*. In Description Logic, this can be a contradiction only if there was a previous assertion that leads to the conclusion that a **Printer** cannot have a *baudRate* property. In the absence of any assertion to the contrary, it is assumed that a **Printer** can have a *baudRate*.

```
Define Concept: Printer ⇒
  (HardCopyOutput
  ∧ ∃printDensity.600DPI
  ∧ ∃paperSize.(US standard ∨ legal)
  ∧ ∃orientation.(portrait ∨ landscape) )
Define Concept: Fax ⇒
  (HardCopyIO
  ∧ ∃paperSize.A10
  ∧ ∃baudrate.64KBS
  ∧ ∃orientation.landscape )
```

Figure 35. Pseudocode for two concepts with properties.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
Define Concept: PrinterANDFax ⇒
   (HardCopyOutput
   ∧ HardCopyIO
   ∧ ∃printDensity.600DPI
   ∧ ∃paperSize.(US standard ∨ legal)
   ∧ ∃orientation.(portrait ∨ landscape)
   ∧ ∃paperSize.A10
   ∧ ∃baudrate.64KBS
   ∧ ∃orientation.landscape )
```

Figure 36. Pseudocode for the definition of the concept (Printer AND Fax).

This result is technically correct, but it is not intuitive that the property *baudRate* should be defined as part of the intersection concept. In fact, this definition seems more like a union than an intersection of the two concepts. However, this anomaly does little harm because these properties do not produce a conflict (unless there is a specific prohibition defined).

The results are more problematic when the concepts to be intersected have a common property, e.g., both **Printer** and **Fax** have a property to define the *paperSize* of the paper. The expansion of the concepts gives a definition of a concept that includes something like *PrinterANDFax* $\Rightarrow$ *(∃paperSize.(US standard* $\vee$ *legal)* $\wedge$ *∃paperSize.A10)*, i.e., a concept with two different *paperSize* properties. This definition is allowed in Description Logic, although the meaning is not intuitive. Furthermore, this conjunction is satisfiable even when *US standard*, *legal* and *A10* are defined to be logically exclusive concepts (e.g., by definition, *paperSize* cannot be both *US standard* and *A10*).

This problem can be addressed by defining an alternative test for compatibility that better reflects an intuitive definition of substitutability. Instead of the conjunction of A and B, the query should define a concept with restrictions that define which properties must be satisfied for the concepts to be compatible. That is, the query should be a definition of "compatible" in Description Logic.

In the example, a test of *substitutability* should be a concept with a property something like *PrinterANDFax* $\Rightarrow$ *∃paperSize.((US standard* $\vee$ *legal)* $\wedge$ *A10)*, i.e., a test for the compatibility of the restrictions on the property. This test can be constructed by defining the query as a new concept, combining the common properties of the two classes, in order to state conditions that must be met by the result.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

This approach is analogous to the intersection operator in relational databases, which is defined to be R – (R – S) (e.g., see [236]). Notably, this test is based on a *closed world model*: for purposes of this test, is it assumed that facts not stated in the definition of the concept must be false.

It is very important to note that this definition comes from an intuition of how to express the *concept of substitutability between two concepts*. The resulting concept is written in Description Logic and tested by attempting to prove that it is satisfiable in the Knowledge Base, *but the test criteria are not derived from Description Logic.*

3.2.2.3. Definition of the Compatibility Test

The section presents the algorithm to test substitutability of two concepts, *C* and *D*. This test defines a logical test, *compatible(C, D),* which is an expression constructed from the definitions of the two concepts. This heuristic uses a non-standard (i.e., outside of Description Logic) interpretation of the concepts. In particular, this test imposes a limited form of a closed world assumption on the two definitions to be tested. However, the resulting concept (the query) is a valid expression in Description Logic, which can be tested against the Knowledge Base.

To test the compatibility of *C* and *D*, an expression is constructed that defines a temporary concept, *CandD*, which has the following definition:

1. Create the concept, *(C AND D)*, i.e., define concept *CandD*: *((definition of C) AND (definition of D))*
2. Find the properties that *C* and *D* have in common
3. For each common property, assert a restriction that combines the restrictions of *C* and *D*.

The temporary concept, *CandD*, is tested for satisfiability.

Table 10. The combination rules for properties. (This table uses the notation from Table 9).

	$\exists.e_2$	**$\forall.e_2$**	**Cardinality**	**Datatype d_2**
$\exists.e_1$	$\exists.(e_1 \wedge e_2)$	$\forall.(e_1 \wedge e_2)$	See Table 11.	FAIL
$\forall.e_1$		$\forall.(e_1 \wedge e_2)$	See Table 11.	FAIL
Cardinality			See Table 11.	FAIL
Datatype d_1				If ($d_1 = d_2$) then { datatype d_1 } else { FAIL }

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

For any property defined by both classes, the restriction is defined to be *compatible* if the conjunction of the fillers is satisfiable. To test this condition, a new restriction is defined that:

1. is the same property
2. combines the type and quantifiers from the two properties (see below)
3. has a filler that is conjunction $(fill_1 \wedge fill_2)$

Table 10 shows the combinations of restrictions with the definition of how they should be combined. Datatypes must match exactly (this is a match of the *type*, not the *value*). Any property of exactly the same type can be combined. The cardinality restrictions can be combined in an intuitive way. Table 11 shows the rules for the cardinalities.

Table 11. The combination rules for quantified properties. (This table uses the notation from Table 9).

	=m.e$_2$	**≤m.e$_2$**	**≥m.e$_2$**
∃.e$_1$	=m.(e$_1$ ∧ e$_2$)	≤m (e$_1$ ∧ e$_2$)	≥m.(e$_1$ ∧ e$_2$)
∀.e$_1$	=m.(e$_1$ ∧ e$_2$)	≤m (e$_1$ ∧ e$_2$)	≥m.(e$_1$ ∧ e$_2$)

=n.e$_1$	If (n = m) {
	=n.(e$_1$ ∧ e$_2$)
	} else {
	FAIL
	}
		If (n ≤ m) {
		=n.(e$_1$ ∧ e$_2$)
		} else {
		FAIL
		}
			If (n ≥ m) {
			=n.(e$_1$ ∧ e$_2$)
			} else {
			FAIL
			}

≤n.e$_1$		≤(max (n, m)).(e$_1$ ∧ e$_2$)	If (n = m) {
			=n.(e$_1$ ∧ e$_2$)
			} else if (n > m) {
			(≥m.e$_2$)
			∧ (≤n.e$_1$)
			} else {
			FAIL
			}

≥n.e$_1$			≥(min(m,n)).(e$_1$ ∧ e$_2$)

In the example above, these results create the desired concept, including the restrictions:

∃paperSize.((US standard ∨ legal) ∧ (A10))

*∃orientation.((portrait ∨ landscape) ∧ (portrait))*Since, *US standard, legal,* and *A10* are defined to be disjoint, there can be no concept in the KB for which *(US standard ∨ legal) ∧ A10)* is **true**, therefore, the compatibility test fails, and **Printer** and **Fax** are not substitutable.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.2.2.4. Summary and Limitations

The substitutability test can be written in Description Logic, as shown in Figure 37. This match is considerably more complex than the other criteria. The *compatible* query is based on an intuitive definition of substitutability, as discussed above.

For Knowledge Base K, with concepts C,

Match (c, d) =

$\{ c, d \in C:$

$(\exists s, s_1, s_2 \in C: d \subseteq s \wedge c \subseteq s$

$\wedge\ parent(c, s_1) \wedge parent(d, s_2)$

$\wedge\ s_2 \subseteq s_1$

$\wedge\ K \models compatible(c,d))$ *("substitutable")*

$\}$

where *parent(c, s)* is defined:

true if $s \neq c \wedge \{\forall t \in C: c \subseteq t \wedge t \subseteq s \Rightarrow c = t \vee t = s\}$

false: otherwise

compatible(c,d) is defined in the previous section.

Figure 37. Definition of the substitutability match.

The heuristic described here applies only to concepts with a single definition of each property. Some classes may have multiple restrictions on the same property. For example, in Figure 38 the first concept defined is a Professor with at least 10 graduate students and also has at least 100 undergraduate students.

The heuristic rule described here cannot be applied to this case. In order to test the compatibility with another concept, it will be necessary to define a heuristic to test the conjunction of the restrictions on the "hasStudent" property. More refined heuristics will need for these more complex cases.

Define Concept: BusyProfessor:
(Professor
$\wedge \geq 10$ hasStudent.Graduate
$\wedge \geq 100$ hasStudent.UnderGraduate)
Define Concept: TeachingProfessor:
(Professor
$\wedge \geq 1$ hasStudent.Student)

Figure 38. A concept with multiple restrictions on the same property.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.2.3. Summary and Formal Statement of the Semantic Match

Figure 39 shows the complete definition of semantic match, stated in Description Logic. The concept matches if any of five conditions is true. Each condition is labeled with a *degree of match*, as in the earlier work.

The *exact*, *plugin*, and *subsume* matches are intuitive and have been discussed above. The *coparent* condition states the intuitive condition that the classes are direct parents. This match finds concepts that may or may not be related to the target concept.

The *substitutable* match is the heuristic defined in section 3.2.2. The definition of substitutable match is more "precise" that the intersection, in that it rejects some matches that would pass the other rule. In the case where $s_1 \equiv s_2$, and *c* and *d* have no properties in common, the substitutability query will give the same result as the intersection query defined by Gonzalez-Castillo, et al. [80] and Li and Horrocks [139].

These match rules are not mutually exclusive; the same concept can match according to more than one rule. By definition, any concept that is equivalent is an *exact* match, must also be a *subsumes* and *plugin* match. The *subsumes* match is a special case of *coparent*, and some concepts may match by *substitutable* and any of the other rules as well.

Figure 40 shows a simple taxonomy graph to illustrate the set of matches. Table 12 gives the results of the matching algorithm for each combination of concepts in Figure 40. In the table, each entry indicates the result of the test *match(row, col)*, i.e., *row* substitutes for *column*. The entry indicates which rule is matched. This table clearly shows the symmetries, and also shows the asymmetric relationship of **C3** and **C4** according to the "substitutable" rule (marked with "*" in Table 12).

The match relation has several simple symmetric relations Table 13 lists the symmetries for the rules defined in Figure 39. In addition, the *substitutable* rule considers more distant concepts; siblings or cousins in the taxonomy graph. These concepts may or may not be substitutable for each other, and this relationship is not necessarily symmetrical.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Definition of Semantic Match

For Knowledge Base K, with concepts C,

Match (c, d) =

$\{ c, d \in C:$

$(c \equiv d)$ *(exact)*

$\vee\ (c \subseteq d)$ *(plugin)*

$\vee\ (d \subseteq c)$ *(subsume)*

$\vee\ (\exists s \in C: parent(c,s) \wedge parent(d,s))$ *(coparent)*

$\vee\ (\exists s, s_1, s_2 \in C:\ d \subseteq s \wedge c \subseteq s$

$\wedge\ parent(c, s_1) \wedge parent(d, s_2)$

$\wedge\ s_2 \subseteq s_1$

$\wedge\ K \models compatible(c,d))$ *(substitutable)*

$\}$

where *parent(c, s)* is defined:

true if $s \neq c \wedge \{\forall t \in C:\ c \subseteq t \wedge t \subseteq s \Rightarrow c = t \vee t = s\}$

false: otherwise

compatible(c,d) is defined in section 3.2.2.

Figure 39. The Definition of Semantic Match.

C1

C6 C2 C3

AND

C4 C5

Figure 40. Simple taxonomy to illustrate the match rules.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 12. The matches for the concepts in Figure 40. (E = Exact, S = Subsumes, P = Plugin, C = Coparent, I = Substitutable, blank= No match). Substitutable Match that is logically consistent is marked with '*'.

	Match (row, column)					
Concept	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**
C1	E	S	S	S	S	
C2	P	E	I*	S	I	C
C3	P	I*	E	I*	S	
C4	P	P	I*	E	I	S
C5	P	I	P	I	E	
C6		C		P		E

Table 13. Symmetries in the Match Relations

A matches B via (exact) ⇔ *B matches A via (exact)*
A matches B via (subsumes) ⇔ *B matches A via (plugin)*
A matches B via (coparents) ⇒ *B matches A via (coparents) or (subsumes)*

4. Summary

Chapter 3 reviewed the basic methods for developing and representing ontologies in a formal logic, Description Logic. This chapter built on this foundation to create two key algorithms needed to maintain and use ontologies in a Ubiquitous Computing Environment.

First, a method was proposed for semi-automatic composition of two ontologies. This algorithm enables a service to dynamically construct a local working set from a large set of ontologies, automatically maintaining logical consistency.

This algorithm is a partial solution to a very difficult problem. The composition algorithm places a few natural restrictions on the design of the ontologies, and requires "hints" from administrators.

A limitation of this algorithm is that there is no inverse operation, i.e., no way to subtract from the working set. If necessary, the Knowledge Base can be flushed, and the working set reconstructed by demand. This is a very inefficient mechanism, though it may suffice for small systems that reboot frequently.

Second, a design and algorithm for semantic query was developed. The architecture defines a general model for advertisement, queries, and query resolution (matching). This algorithm was shown to be an extension and improvement of earlier work.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The query enables applications to frame a generic query that returns a set of concept that are similar to the query according to the current ontology, composed with the first algorithm. These concepts are classes of service; the application must then bind to specific instances of services using conventional mechanisms. The semantic query provides a critical level of indirection, and the discovery of similar concepts should be robust in the dynamic and heterogeneous Ubiquitous Computing Environment.

Chapter 6 presents an implementation of the composition and query algorithms. Chapter 7 evaluates the prototype.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 5. Implementation: Development and Use of Ontologies

1. Introduction

One of the key problems for distributed systems is the creation and encoding of metadata. Chapter 3 presented the foundation of a method: decompose the problem, to create a hierarchy of ontologies. Each ontology represents concepts for a limited *domain*, although concepts can be related to concepts in other domains as needed. Then, the domain ontologies are composed as needed to construct an ontology for a specific environment. The decomposition principle reduces the effort to tractable pieces, while the composition of ontologies enables the sharing and reuse of the pieces.

The process of creating an ontology has two important phases, knowledge acquisition, followed by encoding in a formal language. The knowledge acquisition process is subjective and labor intensive. Once an abstract model is defined, tools assist in the encoding the ontology in a formal language.

This chapter presents a prototype of this approach. In section 2, several example ontologies are developed. Section 3 presents the emerging standards of the Semantic Web, which are used as the formal language for metadata. Section 4 shows how to encode the models described in section 2 in the DAML+OIL XML language.

The ontologies developed in this chapter are input to the prototype Ontology Service described in Chapter 6. The DAML+OIL XML is also compatible with any system that supports the standard.

2. A Model for Real World Objects in a Ubiquitous Computing Environment

Creating an ontology is a process of knowledge capture. As discussed in Chapter 3, the concepts from a model, standard, or informal lore are organized as a hierarchy of concepts, attributes, and relations. This hierarchy is represented in an *ontology,* which must be encoded in a formal language.

Constructing and implementing these models is a fundamental challenge for Ubiquitous Computing. First, there are many environments and applications, so an overall model would be quite complex. Second, different applications and environments have different viewpoints, and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

goals, so models might be quite different. In fact, different tasks or activities might "slice up" the exact same environment in quite different ways. There are many kinds of objects that may be of interest for many kinds of activities in the Ubiquitous Computing Environment. What, then, would be a reasonable set of concepts that are common to all or most smart spaces?

This section develops one example model and several environments in some detail to illustrate the process and provide test cases for prototyping. The models focus on representing and managing physical objects in several environments that might be include in a Ubiquitous Computing Environment. This analysis is important for the design of both system components and the metadata language. For example, the concepts defined by this model can be implemented as proxies for the objects, as discussed in Chapter 3.

In this section, a simple, abstract conceptual model is defined. This model is intended to provide a very high level classification of the objects of interest for many smart spaces. In itself, this framework is too abstract and general to be particularly built. It is intended to be the base on which application-specific models can be derived. For a given application, the concepts of this model will be refined and extended, to create a model for objects of interest for the application or environment.

The limited model is a simple foundation, much more detail would be needed to build a real system. This chapter does not seek to elaborate the abstract model in full detail. Instead, the model is used as an analytic framework to analyze and define fundamental challenges that apply to many specific implementations.

2.1. The Basic Model: People, Places, and Things (PPT)

The first step in the creation of an ontology is to discover or create a logical model of the concepts for the domain. This is an intellectual task, and often is done by a community, e.g., through a standards process. The results might be expressed in natural language, in a formal grammar, as UML diagrams, or in other forms. There are many ways to define the model, depending on the viewpoint of the users and the intended uses.

There is no a priori method to define the categories of such a model, because there are too many different ways to slice up the world. The best that can be done is to select categories to meet a particular goal, in this case, to support a class of applications.

Perhaps more importantly, the purpose of the model is to define objects *of interest to applications*: to model *relevance* as much as objects. The three classes defined here seem clearly

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

be the most interesting for most smart spaces, and are of interest for most if not all smart spaces. These classes provide a balance between simplicity and coverage. For any given application, there may be other, perhaps unique, objects of interest, which must (and can) be modeled. But these other categories are not as universally interesting.

The purpose of this model is not to classify objects in general, it is to classify objects from the point of view of different applications. The criterion of *relevance* means the typology should not be an encyclopedic typology of all possible objects.

However, the model developed here is related to two alternative "upper ontologies". As discussed in section 2.3. The upper ontologies are much more general than the model defined, probably too abstract to be very useful. In contrast, the PPT model is more limited but more useful to applications in a Ubiquitous Computing Environment.

The basic model defines three top level categories (classes) of object: **Person, Place,** and **Thing** (PPT). Table 14 shows the main abstract classes, with basic definitions. **Person** includes both active users and other people, if relevant. **Place** is a physical space that a person could potentially enter and act in. **Thing** is an object (other than a person or place) that a person could perceive and interact with, as defined earlier. (Virtual objects, such as software programs, are not **Things** in this model.)

Table 14. Main Abstract Concepts of the Framework

Concept / Abstract Class	**Definition**
Person	People in the space, both "users" and others
Place	A physical space that people might enter
Thing	A physical object relevant to the application(s)

Clearly, both **People** and **Places** could be considered specific classes or instances of **Things**. However, the **Thing** concept is intended to cover all objects *other than people and places* that are *relevant to the smart space.* Note that this is a use-specific definition: the **Thing** class will be used to model only objects determined to be relevant, and will model them in ways that are relevant to the application(s) to be supported.

Journalists are taught to obtain the context for a story by asking the classic questions "Who, What, Where, Why, When?", and these "five W's" are also used as a definition of "context" in "context-aware" computing (e.g., in [1, 36] and others) and in everyday life (e.g., [84]). The answers to these questions will be in terms of People (Who), Places (Where), Things

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(What). Time (When) and Motivation (Why). This thesis will consider the more static concepts: the base abstractions of the model are "People, Places, and Things" (PPT).

It is not surprising that "People, Places, and Things" have been used in previous systems, notably Taligent [39] and HP CoolTown [130]. Indeed, Taligent claims a trademark on the phrase "People, Places and Things" [225] and patents on related software such as a 'Person' object [49] and a 'Place' object [48], as part of a whole set of Object Oriented frameworks from platform independent operating system, middleware, to applications [39]. The HP CoolTown used the same terms as Taligent, although their approach is conceptually and technically different than Taligent [130].

The Person, Place, and Thing model defined here differ somewhat from Taligent's classes of the same names. Taligent's 'Place' class includes both virtual representatives of physical locations, and purely virtual places, such as bulletin boards ([39], pp. 92-95). Similarly, 'Things' include physical objects, conceptual objects (e.g., work groups), devices, and virtual objects such as screen menus ([39], pp. 95-104). In contrast, in this thesis the concepts are limited to real world objects.

2.2. The Model

This section gives the initial definition of the main objects of the People, Places, and Things (PPT) model, and the model is applied to three example cases. In section 4, the concepts of the PPT model are encoded in DAML+OIL XML.

2.2.1. The Person Object

The **Person** object represents the concepts about a person that are necessary to *link* their physical activities to the virtual world. In a conventional system, people are normally known as either registered *users* (active or not) or unknown. This concept of "user" is too limited for smart spaces. In many cases, it is important to know when one or more people are present, i.e., within some physical area, regardless of whether they are logged in or even have valid system identities.

Figure 41 shows UML for a simple **Person** object. The **Person** object may be related to zero or more **Place** objects (see below). This relationship is labeled "**Person** *isIn* **Place**", with the obvious intuitive intent. The definition of what it means to be "inside" a particular space depends on the space and the application, which is defined in the model for **Place**.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

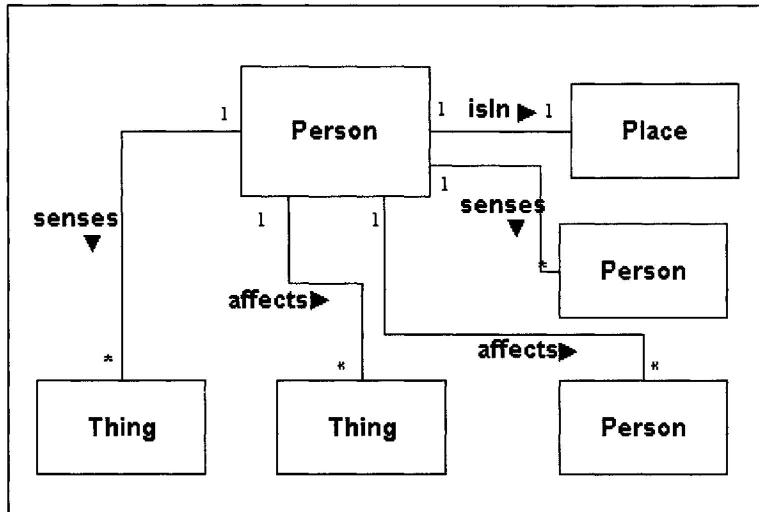


Figure 41. The Person class (from the viewpoint of a single person).

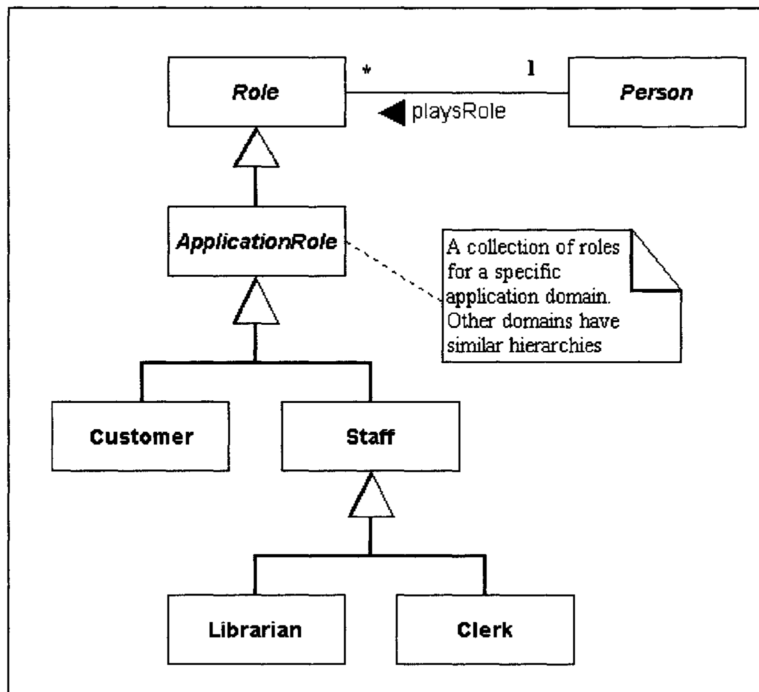


Figure 42. A Person dynamically activates one or more Roles.

The **Person** object may be related to zero or more **Thing** objects (below). Two relationships are described in the UML: *canSense* and *canAffect.* These give an intuitive flavor of the kind of relationship that is conceptually important: the ability to physically interact. Clearly,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

there are many more possible relationships in this vein, although most should be left to specific applications to define.

The **Person** object may also be related to zero or more other **Person** objects. The basic relationships are the same as for a Thing: *canSense* and *canAffect*. This reflects the intuitive fact that people are capable of treating each other as objects. Naturally, for any given application, the human interactions will be an important aspect of the task, and will be modeled with appropriate relationships.

A person is represented by a **Person** object, which adopts (activates) and discards (deactivates) personal roles (**Role**) according to the behavioral context (Figure 42). Each **Role** is a specialized "view", providing an application- or context-specific interface to the **Person**. The **Person** and **Role** classes are developed in detail in other work [148, 149].

2.2.2. The Place object

The **Place** object captures the essential features of a physical location. This is, in fact, extremely simple: a place is something that may contain **Person** and/or **Thing** objects. The place has distinct spatial boundaries, and generally speaking, places are contiguous and compact areas. Containment is meant to be *physical* in some intuitive sense, although the details may differ for different spaces.

Figure 43 shows UML for the **Place** object. A **Place** may be related to one or more **Person** or **Thing** objects, with the relationship "*contains*". The intuitive intention of this relation is obvious, although in some cases the definition and implementation may be subtle. The **Place** class, including a formal spatial model is developed in other work [145, 146].

This simple model of a **Place** neglects relationships between places; e.g., a **Place** composed of several enclosed **Places**. This thesis concerned with objects and people within a single space, so the composition of **Places** is not a central issue. In any case, it could be added to the framework in a straightforward way, if needed.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

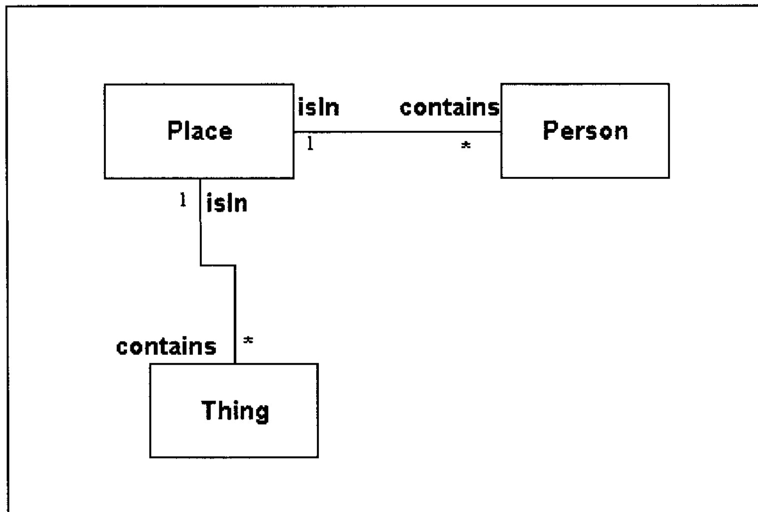


Figure 43. The Place class (from the viewpoint of a single place).

2.2.3. The Thing Object

The **Thing** object encapsulates the basic abstraction for objects other than **Person** and **Place**. As discussed above, this category is very general, and specific sub-classes will be defined for particular applications.

Figure 44 shows UML for the basic **Thing** concept. A **Thing** may be related to zero or more **Place** objects by containment. That is, "Thing *isIn* Place" if and only if "Place *contains* Thing". The **Thing** also has relationships to zero or more **Person** objects, reflecting the reciprocal of the relations described above: *isSensedBy* and *isAffectedBy*.

An important service of the **Thing** object is to provide a description of the (actual physical) object. This information is made available through services such as registries, brokers, and databases. The descriptive information is part of the metadata of the system.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

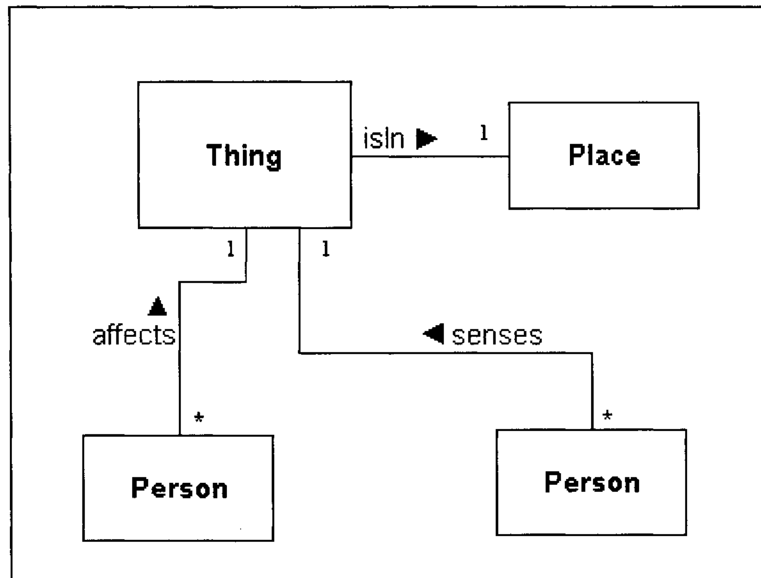


Figure 44. The Thing class.

2.3. Grounding in An Upper Ontology

Chapter 3 argued that the concepts of the models should be mapped to an all-encompassing *upper ontology*, capable of expressing all concepts of interest to all domains and sub-domains. This study tentatively adopts two upper ontologies that seem to have growing support and use. To illustrate this principle, the concepts for the "Person, Places, and Things" (PPT) model will be mapped to these two upper ontologies.

2.3.1. Two Upper Ontologies

One ontology that is particularly applicable for Ubiquitous Computing is the Basic Formal Ontology (BFO) [15]. The BFO is a conceptual ontology intended to give a unified description of spatial-temporal concepts of physical reality, which includes the concepts relevant to Ubiquitous Computing Environment.

The BFO defines two fundamental views of reality: three-dimensional (termed *continuent*) and four-dimensional (termed *occurrent*) [82]. The three-dimensional approach views the world as snapshots of successive instants, while the four-dimensional approach views the world as space-time worms (nicknamed "SNAP" and "SPAN", respectively). This work is particularly noteworthy because these two different viewpoints reflect critical issues of persistent identity and temporal flow. The BFO specifically considers how to map between these fundamentally different viewpoints via a "SNAP-SPAN Trans-ontology", which defines a formal

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

theory to relate objects to their history. This theory provides an important framework for creating a comprehensive model for the Ubiquitous Computing Environment.

The second upper ontology is an attempt to unify many existing taxonomies. In recent years, researchers and standards organizations have dedicated significant effort to developing comprehensive standard ontologies for use in computer and information systems. In some cases, the ontology is intended to capture "common sense" (i.e., natural usage), e.g., CYC [182]. In other cases, the goal is linguistic translation, e.g., EuroWordNet [54]. And in other cases, the goal is to unify multiple standard vocabularies, e.g., Ontomap [179].

The IEEE P1600.1 Standard Upper Ontology working group is developing a consensus ontology intended to unify many of these efforts with a single ontology [121, 167]. The Standard Upper Merged Ontology (SUMO) will contain more than 1000 terms and definitions and will be expressed in several encodings, including XML, e.g., as OWL.

The SUMO builds on previous work, it is the result of a broad consensus, and it has a formal definition. For these reasons, the SUMO is likely to become widely accepted as an upper ontology for computer systems, including Ubiquitous Computing Environments.

The SUMO activity also encompasses "Mid-Level Ontology" (MILO) which is intended to bridge between the upper ontology and domain ontologies. The MILO defines many useful "utility" concepts, thought to be useful for the definition of domain ontologies. SUMO also is developing high-level domain ontologies, e.g., for financial concepts, and Quality of Service [121].

In section 2.3.3, the top level concepts of the People, Places, and Things model are defined in terms from the BFO and SUMO ontologies.

2.3.2. Grounding the PPT Model in an Upper Ontology

The top concepts should correspond to concepts in an upper ontology. This section shows how they should be defined in the BFO and SUMO.

2.3.2.1. Defining the PPT using the BFO

The PPT model is three-dimensional (i.e., it is an object model, not a process model), so all three concepts are instances of BFO's **Continuent** (SNAP). The PPT concepts can tentatively be defined to correspond to concepts in the SNAPBFO. "Place" is a "3-dimensional Spatial Region", a "Person" is an instance of "Substantial Entity", and "Thing" is the class of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

"Substantial Entity". Table 15 lists the hierarchical classifications based on the definitions of BFO [15].

Table 15. Provisional correspondence of PPT concepts and BFO "SNAP" ontology.

PPT Concept	BFO SNAP Concept
Person	Continuent::Independent Entitiy::Substantial Entity::Substance::Organism
Place	Continuent::Spatial Region::3-dimensional
Thing	Continuent::Independent Entity::Substantial Entity

The mapping of the PPT to the BFO is fairly simple. The exercise generates several potentially important insights into the PPT model.

First, the BFO highlights an essential design decision of the PPT, namely, it is conceived as an object model—a three-dimensional snapshot of identifiable entities (BFO *continuents*). Clearly, there are many **Occurrent** concepts that are important in Ubiquitous Computing Environments, including processes, sessions, and temporal aspects of policies. It should be recognized that these concepts are outside the PPT model.

Second, the BFO (and many other abstract systems) would consider "Place" and "Person" as sub-classes of "Thing". The PPT model specifically defines "Thing" to exclude these classes, as well as excluding computers and software. Therefore, the PPT category called "Thing" is a rather complex sub-set of the "things" in the BFO ontology, and does not fit well at the high level of the upper ontology. While this concept is complicated, it expresses a fundamental theoretical statement about what is important to Ubiquitous Computing Environments.

2.3.2.2. Defining the PPT using the SUMO

The concepts of the PPT map to concepts of the SUMO: the "Person" concept corresponds to "Human", "Place" corresponds to "GeographicArea", and "Thing" corresponds to a sub-class of "Object". Table 16 lists the correspondences.

Table 16. Provisional correspondence of PPT and SUMO.

PPT Concept	SUMO Concept
Person	Human
Place	GeographicArea
Thing	Object

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

An important feature of SUMO is that it is already defined in a DAML+OIL XML ontology. Therefore, the correspondences described here can be directly implemented as DAML Axioms. For example, we could define:

sameClassAs(ppt.daml#Person, sumo.daml#Human);

The SUMO hierarchy can generate additional insights into the PPT model.

Figure 45 shows the SUMO definition of "Human", which corresponds to "Person". The definition is quite elaborate, and contains many facts not likely to be useful in a UCE (e.g., Human *isA* Hominid *isA* Primate). On the other hand, the superclasses for "Agent" are quite pertinent to UCE: there will be many cases where Humans and software Agents should be considered related or equivalent concepts.

Figure 46 and Figure 47 show the definition of "GeographicalArea" and "Object" respectively. The SUMO does not develop spatial concepts in much detail, compared to either BFO or the PPT model. As in the case of the BFO, the PPT concept "Thing" corresponds to a very specific subclass of "Object".

2.3.2.3. Discussion

The mappings to upper ontologies may ultimately yield theoretical insight and formal clarity for the PPT model. Furthermore, the upper ontologies offer a route to composing or unifying with other theories, e.g., based on temporal logic.

On the other hand, most of the information in the upper ontology has little practical value to the Ubiquitous Computing Environment. For example, the fact that a person is a *primate* and a *hominid* will rarely be salient to a Ubiquitous Computing Environment. This should not be a surprise: the PPT concepts were selected to reflect concepts relevant to Ubiquitous Computing Environments, they would be expected to be a preferred terminology.

In summary, while an upper ontology may yield valuable insight, it is not necessarily useful for an application in practice. Upper Ontologies serve to assure theoretical correctness and trans-ontology mappings, but do not necessarily aid specific tasks.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

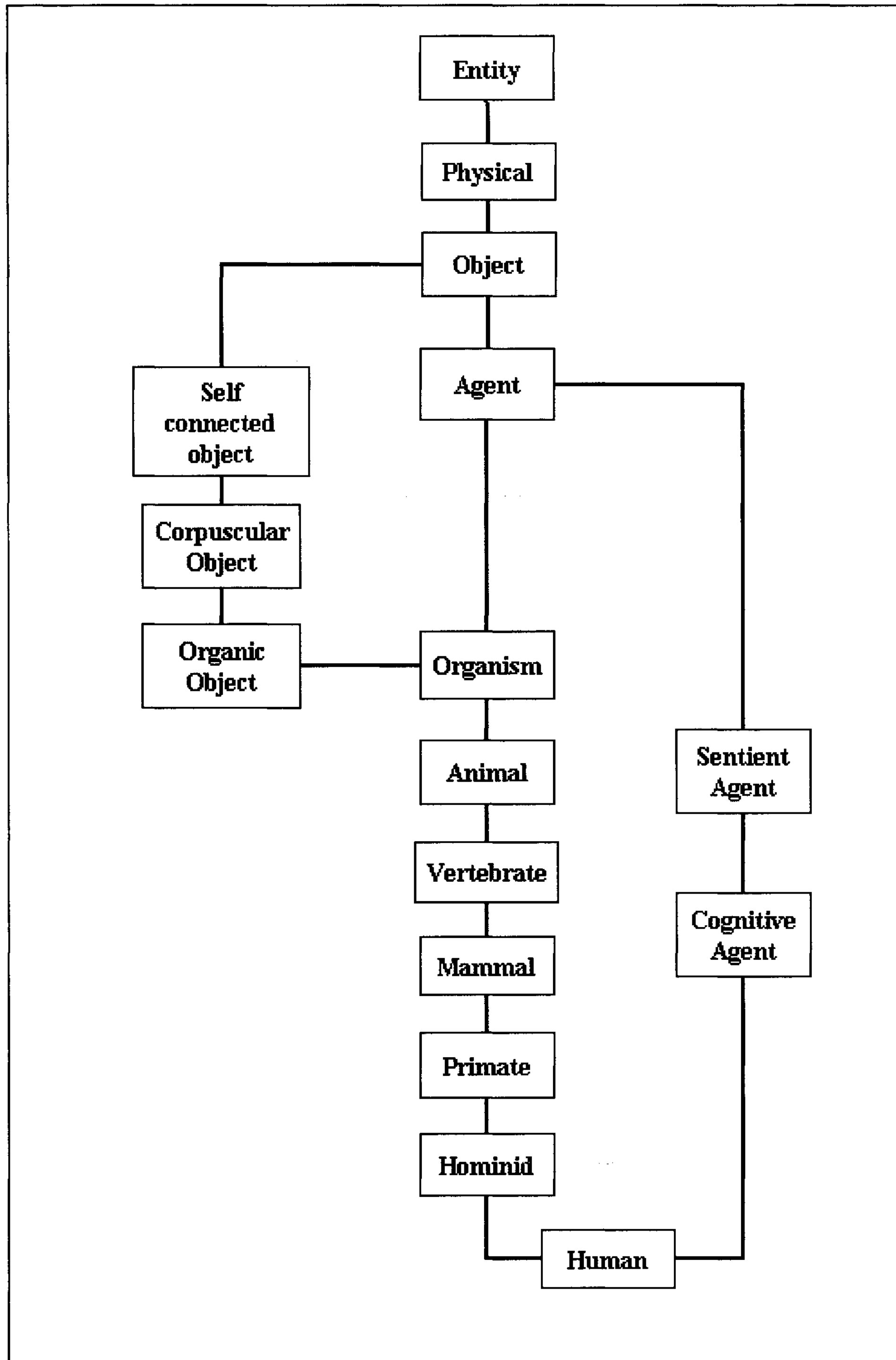


Figure 45. The SUMO hierarchy for the concept "Human".

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

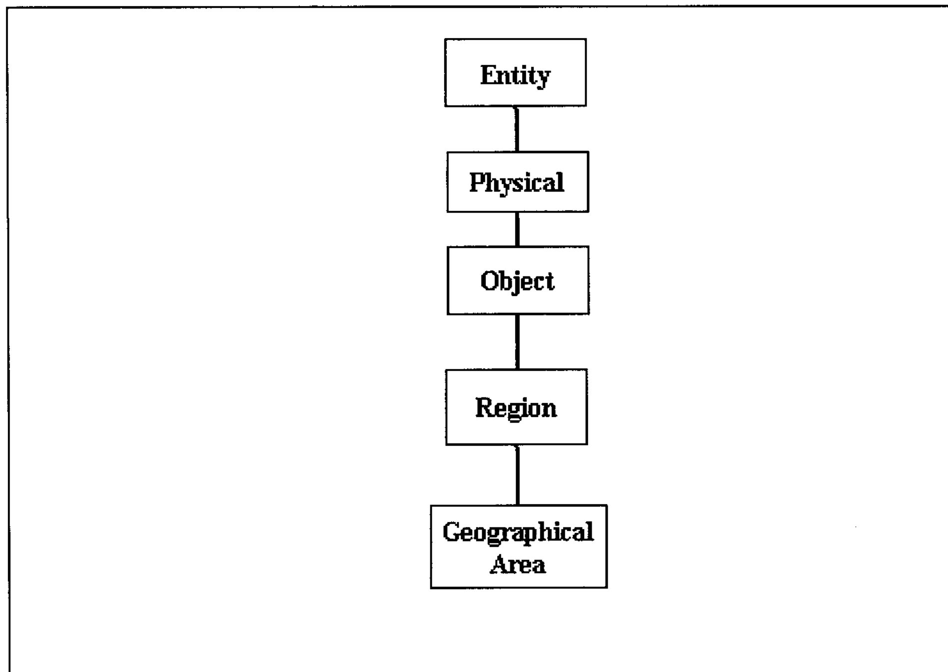


Figure 46. The SUMO hierarchy for the concept "Geographical Area".

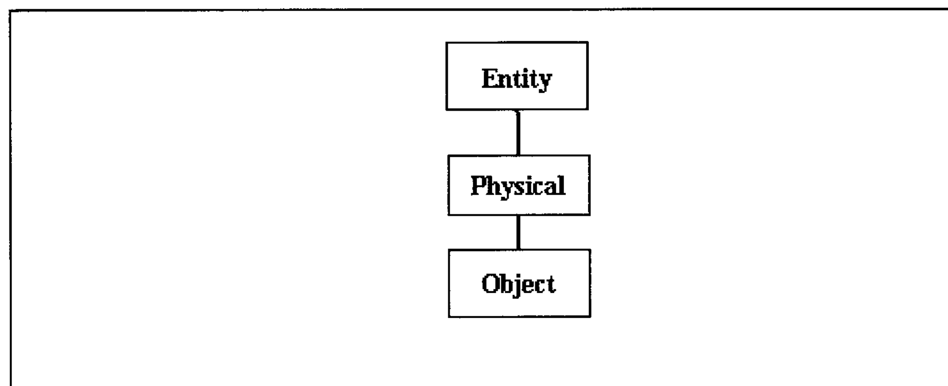


Figure 47. The SUMO hierarchy for the concept "Object".

2.4. Use of the Abstract Classes of the PPT Model

The abstract classes defined above are intended to provide the foundation for creating appropriate proxy objects and metadata records. As discussed in Chapter 3, proxies for the **Person, Place**, and **Thing** hide the complex and dynamic infrastructure, and provide useful abstractions for the applications.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The model for a particular space must represent all the concepts needed by the dynamic set of users and applications of the space. The model for a specific space may be composed from several specialized application models.

Different applications will need to use different real world objects, and will need different attributes for different interactions with the same given object. For example, a bookstore will view an individual book as an instance of a class of "merchandise", with properties such as name, units on hand, retail price, and supplier. A library views the same book as an instance of "information resource", with attributes such as authority, description, references from, references to, and possibly a more detailed view of the contents in the book.

Since these views are essential to the applications, there is no question of avoiding or eliminating them, or of imposing a single, all-purpose view on the system. This is neither feasible (the number of views is infinite, it is difficult to capture the application's view, views change) nor desirable.

The goal is to define a hierarchy of abstractions that capture useful levels of generality across many kinds (and views) of objects; abstractions which can be specialized for specific applications. To continue the example above, the authority and description of the book is common to both a book store and a library (although perhaps used for different purposes), and can be captured in an abstract class for a **Book**, a sub-class of **Thing**.

However, aspects of the same book could be modeled quite differently for a store than for a library. In a store, a book is treated the same as any other good for sale. It has the same properties and operations as a loaf of bread or a shirt: it has an inventory record, identified by a UPC, it has a price, and it can be sold to a customer. Not much else about a book matters to the store.

In a library, the same book has a much more complex existence, and there are many recognized sub-categories of **Book**. For example, the book may be indexed by a variety of attributes (but not usually UPC or price), and the operations do not include *purchase*, but do include *browsing*, *borrowing*, and *reserving*. Library services may well "point inside" the content of a single book, recognizing specific chapters or passages as objects of interest. Thus, the exact same physical object may have a substantially different representation in two different spaces.

To illustrate this approach, three example applications are briefly presented here:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- "Shopping Mall" – a place for retail shopping
- "Hospital" – a place dedicated to delivery of health care services
- "Library" – a place dedicated to providing access to an organized collection of information

These three environments have substantially different purposes and activities, but share the important features defined above. They are complex spaces in which people interact with both physical objects and information systems, and the interaction is essential to the purpose of the space. Each environment is designed to support the specific activity, and there are large amounts of inventory and other on-line information about the physical objects.

2.4.1. "Shopping Mall" – A Retail Sales Space

A "Shopping Mall" is a space designed to sell merchandise. Many retail malls also provide a variety of other attractions, such as play areas, community displays, theaters, and also may include office space, medical facilities, and public services. This section will consider only the retail sales functions of the "Shopping Mall" space, which are common to a great variety of spaces, from single small stores to large mega-malls; spaces which may not share other attributes except for their mercantile purpose.

While there are many variations on the design, the space is usually a controlled public space, in which users may enter and move easily, with strong authentication required for transactions (sales). A large part of the space is dedicated to commercial messages in many forms, designed to attract customers to specific goods and services.

A "smart" space for a shopping mall might provide many services, including:

- Guides and maps, which give the location of shops and other facilities.
- People locator, both "you are here", and finding others, such as children.
- "Broadcast" and "narrowcast" public announcements, customized to the person and their location
- Search services, to locate merchandise or services of interest
- Access to financial services, e.g., to access personal accounts to arrange payment

These services are not especially challenging or novel in themselves, but they must deal with a very large number of people and objects of interest, in a public (but controlled) environment. And while the environment is controlled, it consists of many autonomous entities,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

including individuals and enterprises, which must share and interact in standard and controlled protocols.

Some of the objects of interest in this scenario are shown in Table 17.

Table 17. Some objects of interest for the "Shopping Mall"

Class	Description
Person	Sales Staff, Customers
Place	Store, hall, parking lot
Thing	Goods, elevators, signs, ATMs

2.4.2. "Hospital" – A Healthcare Delivery Space

In this section, a "Hospital" is defined to be an environment for delivery of medical and health services.* There are a variety of similar environments, including clinics, doctor's offices, large and small hospitals. These environments are highly controlled; all objects and people are inventoried upon entry and tracked through the system. This tracking serves many purposes:

- medical personnel and equipment are tracked to assure availability in critical need
- materials are tracked for accountability and safety
- patients are tracked for safety and quality assurance
- activities are carefully logged for accountability, safety, and continuity

Contemporary Healthcare systems are extremely complex, and have elaborate information systems. (See, perhaps, [72], Chapter 3.) A large-scale healthcare institution, such as a major clinic or hospital, is actually a multipurpose space, encompassing a large variety of specialized areas, including waiting areas, examination rooms, dormitory rooms, laboratories, and operating rooms—as well as areas dedicated to food service, retails sales, and financial services. A full analysis of even a single example of a healthcare environment—even a single facility, such as a medical laboratory—is far beyond the scope of this project.

---

* Disclaimer: The purpose of this section is to illustrate the general applicability of the model to this application. This discussion is necessarily superficial and should be regarded as speculative. No claim is made that this model or prototype is suitable for use in any specific Healthcare applications.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

For example, consider a limited example of a healthcare space, such as a relatively small clinic, and the just activity related to a "patient visit": tracking the activities of a patient as he enters the space, interacts with the staff and objects, and moves throughout. It should be clear that this activity has many similarities to the Shopping Mall discussed above.

Table 18 lists the main classes of objects for the "Hospital".

Table 18. Come objects of interest for the "Hospital".

Class	Description
Person	Staff, Patients
Place	Rooms and areas
Thing	Medical materials, equipment

2.4.3. "Library" – A Space to Access An Information Collection

A library can be seen as an environment for locating and accessing information. In general, a "library" is an *organized collection of information*, designed to enable users (patrons) to discover information relevant to some question or task. The library owns rights to a collection of resources, both physical (e.g., books) and digital (e.g., databases), and infrastructure for locating and accessing these resources.

One of the key user activities in a library is *seeking information*. The user has one or more questions or topics, and is seeking information relevant to his or her current interest. In the library, the user presents his question(s), discovers resources that may be relevant to his interests, accesses resources to evaluate their usefulness, and obtains copies of selected information.

Much of the library's infrastructure consists of tools and support for finding information relevant to the user's goals; i.e., for relating user questions to sets of resources available in or via the library. These tools include indexes, guides, manuals, reference works, and librarians. In addition, the collection itself may be organized to facilitate browsing, e.g., with topically related materials grouped together on open shelves.

Some of the objects of interest in the library are shown in Table 19.

Table 19. Some objects of interest for the "Library"

Class	Description
Person	Library staff, patrons (users)
Place	Branch library, Floor, shelf location
Thing	Resources (books, etc.), shelves, copiers, etc.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3. Metadata Encoding Using the Semantic Web Standards

Section 2 defined a conceptual model. The next step is to extract and encode the concepts from these models in several formal ontologies. This section presents the standards of the Semantic Web. The following sections show how to encode the concepts of the model using the standards of the Semantic Web.

The "Semantic Web" is an emerging suite of XML-based standards [14]. The Extensible Markup Language (XML) with XMLSchema has become the encoding standard of choice for metadata [239, 241, 242, 245]. XML provides a standard for structured or semi-structured encoding of data, along with conventions to address the important problems of name spaces, references (Universal Resource Identifiers, URI), and standard data types. XML is easy to use and can be used as a lingua franca between different representations, e.g., as ingest to an SQL database or as a message format as in SOAP [248].

However, the XML standard only defines syntax: there are an infinite number of languages that may be expressed in XML, and each XML document may have many different interpretations. The Semantic Web addresses this problem with XML languages that are encodings of formal languages.

Unlike most XML languages, the Semantic Web languages are defined with a formal semantics, i.e., each XML statement maps to a statement in a formal logic. The semantics assures that two independent parties can exchange XML documents with some assurance that they "mean" the same thing. The mapping also enables automated reasoning to be used, to prove consistency and discover relations.

The Semantic Web XML languages can be mapped to different logical systems, and the automated reasoning can be implemented by many methods. This thesis focuses on a particular subset of first order logic, called *Description Logic*, which provides expressive yet computationally efficient languages for classification [81]. However, the Semantic Web languages can be mapped to other logics, such as F-logic [128]. Likewise, alternative implementations of automated reasoning exist, and the Semantic Web can be implemented by multiple technologies.

This study uses the DAML+OIL XML language to express metadata about ubiquitous computing environments. DAML+OIL and similar languages (OIL [59], KIF [71], OWL [252])

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

define a formal semantics for statements. Specifically, all the statements of DAML+OIL are mapped to statements in Description Logic [63, 100], as described in Chapter 3. The mapping to Description Logic means that DAML+OIL documents are expressive and can be analyzed by computationally tractable algorithms.

3.1. The Semantic Web Stack

The World Wide Web Consortium (W3C) defines a "stack" of Semantic Web standards (Figure 48). The World Wide Web standards provide a universal address space (Universal Resource Identifiers (URI) [13]), and the Extensible Markup Language (XML) language is a universal standard for content markup, which is a universal (and multilingual) syntax for structured text ([239, 241, 242, 245]). An XML document is guaranteed to be parseable, in that the tokens and the structure of the file can be determined from the standard. However, there is no constraint on how to interpret the tokens: the same information can be encoded in many ways using XML.

The Resource Description Framework (RDF) [240] and Resource Description Framework Schema (RDFS) [244] defines an XML language for stating facts and defining entity-relationship diagrams. Essentially, RDF defines standard XML tags for stating the entities and relationships in a network of related objects.

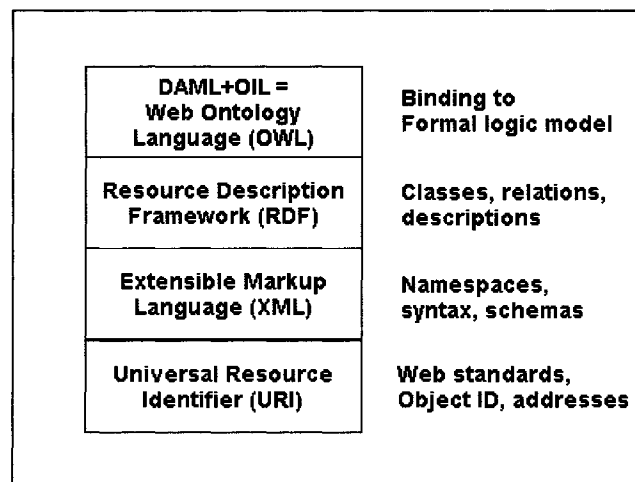


Figure 48. The Semantic Web Stack.

The original RDF/RDFS specification did not specify a single logical model of entities or relationships: the same relationship could be encoded in many ways. For example, Figure 49a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

and b show two of many possible RDF models for the same concept, "Elena is a friend of Bob". The RDF language specifies the standard for encoding these graphs in XML, but RDF does not specify how to interpret the nodes and arcs. Furthermore, id may be difficult to determine if the diagram in Figure 49c is the same or different as Figure 49b. (That is, are 'isFriendOf' and 'isAnnoyedBy' equivalent?)

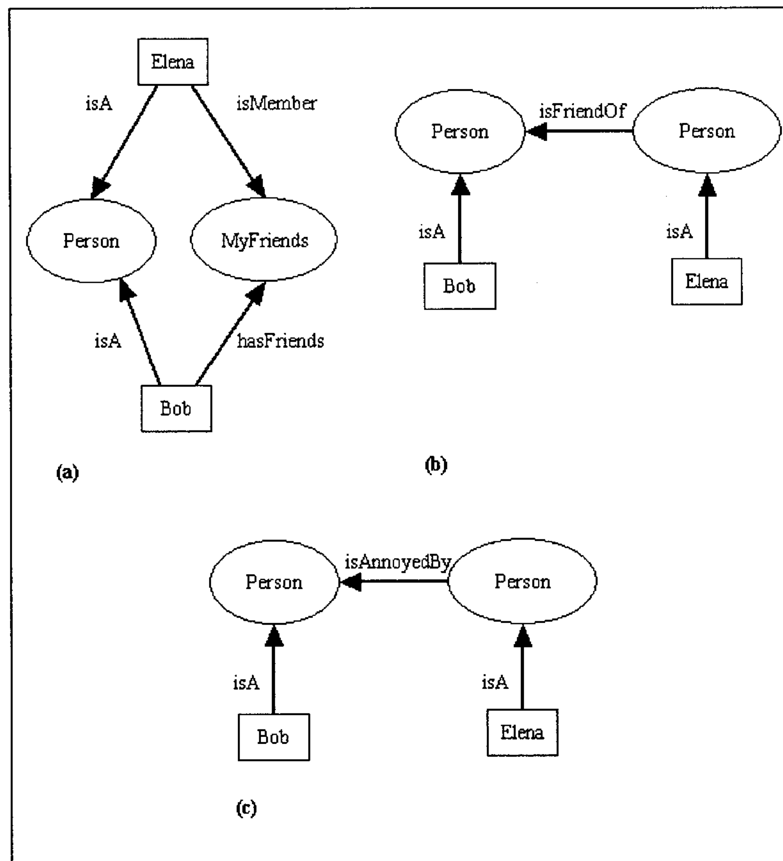


Figure 49. Two different RDF definitions (a), (b), of the same concept (follows the common RDF notation, e.g., [243]), and a third concept, (c), indistinguishable from (b).

In recent work, RDF has been put on firmer ground by defining a model theory and standard for inference (e.g., see [103, 115]). This work provides a sound foundation for defining additional languages for ontologies on top of the facilities provided by RDF.

The DARPA Agent Markup Language (DAML) and Ontology Interchange Language (OIL) are XML languages (combined as DAML+OIL) were an initial effort to design an XML language for Description Logic. The OIL is a language for describing formal vocabularies (i.e.,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ontologies): essentially a meta-format for schemas [59, 110, 111]. The DAML is a language for describing entity-relationship diagrams that conform to a schema (i.e., an OIL ontology) [7, 100-102].

The DAML+OIL language is an XML binding to a formal logical model. Specifically, each XML tag represents a corresponding statement in Description Logic. This means that a DAML+OIL XML file can be translated into a set of statements in Description Logic, and Description Logic proofs can be applied to DAML+OIL XML.

Table 20 gives a summary the logical concepts and the DAML tags that represent them (adapted from [80, 204]) For alternative statements of the formal semantics, see also [63, 100-102].

Essentially the DAML+OIL language uses the mechanisms of XML to deliver *well-defined logic programs*. Therefore, unlike XML and RDF alone, a DAML+OIL document has a single, universal interpretation. While there may be many ways to express the same idea in DAML+OIL, a given DAML+OIL document has only one correct interpretation.

This language extends the capabilities of RDF in important ways. For example, in the example in Figure 49, additional constraints can be stated, to flesh out the intended meaning. These might include:

- "MyFriends" is the class of Person such that isFriendOf.Person.
- "MyPests" is the class of Person such that isAnnoyedBy.Person.
- "MyFriends" and "MyPests" are disjoint classes.
- "Elena" and "Bob" are disjoint individuals

The DAML+OIL language was has led to the specification of the Web Ontology Language (OWL) as a Web standard [246, 247, 249, 252]. Like DAML+OIL and other earlier work, OWL is an XML language with a formal semantics and mapping to Description Logic [115]. OWL extends RDF/RDFS to provide important features, including:

- ability to declare classes as logical combinations (intersection, union, complement) of classes
- declare a property to be transitive, symmetric, functional, or the inverse of another property
- use datatypes (such as integer and boolean)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 20. Correspondence of Description Logic and DAML+OIL (see also [63, 80, 100, 102, 204]). (Concepts A, C, D; Roles R, S; type T, U; instance o, p, d)

DL Expressiveness	DL Syntax	DAML/XML Syntax
$\mathcal{ALC}$, also called $\mathcal{S}$ when transitively closed primitive roles are included	A	daml:Class
	$\top$	daml:Thing
	$\bot$	daml:Nothing
	$(C \subseteq D)$	daml:subClassOf
	$(C \equiv D)$	daml:sameClassAs
	R	daml:Property
	R	daml:ObjectProperty
	$(C \cap D)$	daml:intersectionOf
	$(C \cup D)$	daml:disjunctionOf
	$\neg C$	daml:complementOf
	$\forall R.C$	daml:toClass
	$\exists R.C$	daml:hasClass
$\mathcal{N}$	$\leq nR.\top$	daml:maxCardinality
	$\geq nR.\top$	daml:minCardinality
	$= nR.\top$	daml:cardinality
$\mathcal{Q}$	$\leq nR.C$	daml:hasClassQ
		daml:maxCardinalityQ
	$\geq nR.C$	daml:hasClassQ
		daml:minCardinalityQ
	$= nR.C$	daml:hasClassQ
		daml:cardinalityQ
$\mathcal{I}$	R^{-}	daml:inverseOf
$\mathcal{H}$	$(R \subseteq S)$	daml:subPropertyOf
	$(R \equiv S)$	daml:samePropertyOf
$\mathcal{O}$	$\{o, p, \ldots\}$	XML Schema (XMLS) Type + rdf:value
	$\exists T.\{o, p, \ldots\}$	daml:hasValue
$(\mathcal{D})$	U	daml:Datatype + XMLS Type
	T	daml:datatypeProperty
	$\exists T.d$	daml:hasClass + XMLS Type
	$\forall T.d$	daml:toClass + XMLS Type

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

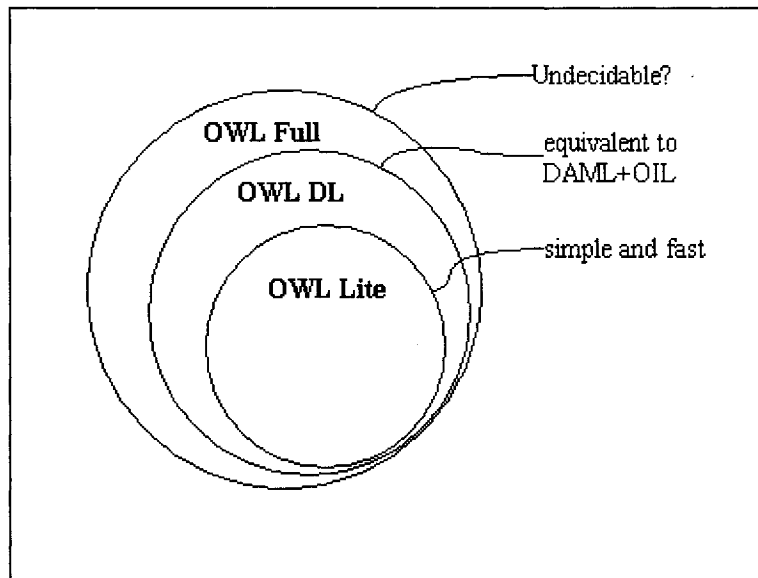


Figure 50. The subsets of the OWL language.

The OWL standard defines three related languages (Figure 50). "OWL Full" is a general language, capable of expressing a large subset of First Order Logic. The formal properties of OWL Full are not fully understood, although it contains constructs for which inference is known to be undecidable [113]. The W3C has defined two subsets of the full language with substantially different expressiveness, decidability, and computational complexity. "OWL DL" is a subset of OWL Full that maps to a Description Logic. OWL DL is similar to DAML+OIL with only minor changes [113]. "OWL Lite" is a restricted subset which maps to a simple but very efficient Description Logic [113].

DAML+OIL and OWL DL provide languages that enables Description Logic reasoning to be used in the World Wide Web. These languages use XML syntax for exchange, URI references for names, XML schema datatypes, and the ability to publish and retrieve information using the Internet. This thesis shows how these features solve key problems for Ubiquitous Computing Systems, as well. This study used the DAML+OIL, but the ontologies will be converted to OWL in future versions.

3.2. Encoding the Ontology in XML

The concepts of the ontology must be encoded in DAML+OIL XML [42] (DAML). Using a tool such as OilEd [8, 178], the concepts are defined step by step.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In the terminology of DAML, each *concept* in the ontology is represented by a DAML *Class*. The *attributes* are represented as DAML *Properties*. Properties are relations, they have domains and ranges. The attributes of a concept are represented as a *Class* with a *Restriction*, i.e., the class is defined to have the specified *Property*.

The concepts of the ontology are related as a hierarchy by logical subsumption. This is encoded in DAML as *subclassOf* relations. So, to represent the concept "C isA D", *C* and *D* would both be a DAML *Class*, and *C* would have the property *subclassOf D*.

A class can be defined as a combination of other classes. To state that "A isA (B AND C)", *A* is defined as *subclassOf A* and also *subclassOf B*. A class may also be a restricted subclass of another. For example, to say something like *A* is a *B*, where *B* must have the property *X*, the ontology could define *A* to be *subclassOf* (*B hasClass X*).

The DAML language also defines a limited set of Axioms: *disjoint(A,B), sameClassAs(A,B), subclassOf(A,B)*. These axioms can be used to directly state constraints and relations in the ontology. They can also be used to state relations between concepts in two different DAML ontology documents, e.g., to define corresponding terms from two different domain specific terminologies. This mechanism can be used to compose ontologies, as discussed in section 6 below.

An ontology development tool will have the ability to validate the ontology under construction. This is done by encoding the classes that have been defined as a series of assertions to a Knowledge Base (KB), and then proving that the KB is logically consistent. The ontology is proved correct by proving that every class (concept) is satisfiable in the KB, i.e., that there is no logical contradiction in any of the definitions. This proof may also discover implicit subsumption and equivalence, i.e., relations that were not explicitly defined.

For example, the OilEd tool has several reasoning engines that may be selected [8, 178], and other tools have similar choices. The classes of the ontology are encoded into statements in the logic of the specific Knowledge Base. If contradictions are discovered in the KB, the OilEd tool will indicate classes and properties that are invalid.

When all the concepts have been entered and the ontology is validated, the ontology can be written out. Most tools offer several alternatives for writing the ontology, including DAML+OIL and OWL XML, as well as native formats for specific tools and Knowledge Bases.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The OilEd tool has options to write the ontology in several forms including DAML+OIL and OWL XML documents. Once the ontology is proved to be valid, the resulting DAML+OIL is correct, i.e., contains no logical contradictions. The automated tools manage technical details of the XML, especially, XML namespaces. Each object in the ontology has a unique name based on the XML namespace (URL) of the ontology document. The OilEd tool automatically generates these names when needed. A DAML ontology may import (reuse) objects from other DAML ontologies. This is done by importing an XML namespace, and referring to the tag in that namespace. The OilEd tool manages this operation as well.

3.3. Heuristics for Knowledge Capture

Whatever the process and format, the ontology must define a hierarchy of concepts, attributes and restrictions on each concept, and relationships between concepts. This section identifies several heuristics that guide these definitions.

As discussed above, a concept corresponds to a taxonomic classification, a set of "*isA*" relationship. In a object model (e.g., expressed in UML) there is a natural correspondence that stems from the underlying logic of the inheritance hierarchy. The *classes* are the prime candidates to be the *concepts* of the ontology. The *attributes* of a UML class will be modeled as *properties* in the ontology.

However, it is not sufficient to just translate classes to concepts. Object models represent classification information in several ways besides class definitions. For instance, classification is sometimes represented as a variable or a combination of several variables that partition objects into classes. In designing an ontology, there is a choice whether and how to represent this classification.

For example, a model of devices might define a single class, "Device", with variables (attributes) called "doesInput" and "doesOutput". (Figure 51a) Depending on the value of the variable, this single class represents four categories of device: "no I/O", "input only", "output only", and "both input and output". In this case, this classification might be represented by three concepts, "InDevice", "OutDevice", and "InOutDevice". The latter is defined as "InDevice AND OutDevice". Figure 51b sketches this alternative.

Representing the classification as concepts makes it easier to determine important relationships among the device classes. For instance, it may be possible to determine that "InDevice" does not match "OutDevice" or "InOutDevice", while "InOutDevice" might match

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(substitute for) “InDevice”. These relationships are not necessarily apparent from the four-fold classification using two variables.

In principle, any variable can be treated as a set of categories. But judgment must be applied to decide what categories should be defined. For example, creating a classification from a variable (or combination of variables) with a large (or infinite) number of values will create a disastrous number of categories, so it is usually preferable to define a reasonable number of ranges. For example, the age of a person might be divided into a few relevant ranges, such as “under 18”, “18-65”, “over 65”.

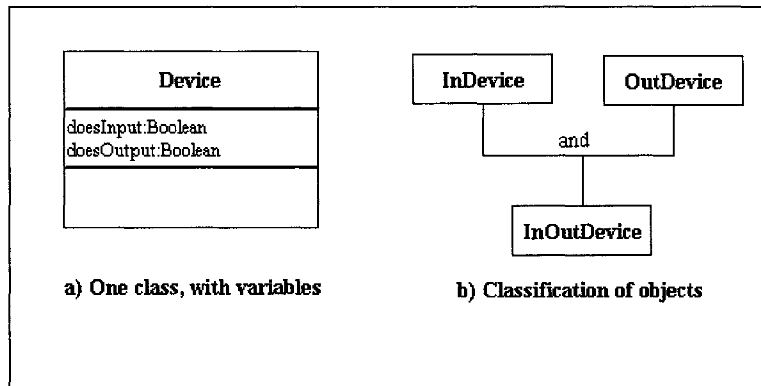


Figure 51. Variables may represent classifications.

It should be clear that the same real world entities might be classified in different ways depending on the purpose of the ontology. In fact, this is the essence of good design: the ontology should represent the concepts that are important for the application that will use it. This diversity conceptual views is the fundamental source of the Tower of Babel problem, which ontologies seek to solve.

In addition to encoding the class hierarchy, the ontology can define logical constraints. Axioms can define relationships (such as equivalence), logical constraints such as disjunctions (e.g., disjoint(MALE, FEMALE)), and abstract concepts (e.g., SOCCERMOM = ($\geq$ 1 CHILD $\wedge$ $\forall$drives.Van)). These relationships are not represented in the UML Entity Relationship diagram.

3.4. Summary Deployment and Usage

Chapters 2 and 3 described a general approach to managing metadata for Ubiquitous Computing Environments This approach can be implemented using ontologies encoded as XML

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

documents. Ontologies can be developed and published at well-known URLs. Upper ontologies will be developed and maintained by international standards bodies, such as the IEEE. Domain ontologies will be developed by industry and discipline organizations. Specialized ontologies can be developed by any group or community that needs one.

XML name spaces and URIs are used as the mechanism to relate specific terms (concepts) to one or more ontologies in which it is defined. Each concept is identified by a unique name within the ontology, and each ontology can be identified by its URL.

Users and applications can obtain ontologies as needed by loading documents from the specified URLs. For instance, in a message exchange, the sender and receiver can coordinate by agreeing to use a specific ontology available from the network. Note that a given application needs to download only the ontology documents it needs. There is no need for a single, centralized ontology.

4. Examples of DAML+OIL Ontologies

In this section, the Person, Place, and Thing (PPT) model is used to create a top-level ontology, encoded in DAML+OIL.

Two example domain ontologies are developed, one for Library Resources (e.g., books, articles, etc.), and the other for devices that might be used in a Ubiquitous Computing Environment, such as printers. Each ontology is developed in detail and encoded in DAML+OIL XML. Certain classes in the domain ontologies are identified to be related to classes in the other ontologies. For example, "LibraryResource" is defined as a sub-class of "Thing". These trans-ontology relations are applied when the ontologies are composed.

The two domain ontologies illustrate the method for knowledge acquisition and encoding. The Library Resources ontology used standards developed by domain experts. The ontology was constructed from a close reading of the BNF grammar. The Device ontology represents knowledge abstracted from a software design.

Each ontology was developed as a self-contained model. The individual ontology was not constrained by other domains, and the developers only needed to know about the domain of interest. It was only necessary to consider the concepts and relations relevant to books and articles, or devices, and so on, employing relevant standards, if available.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In this study, the OilEd tool was used to construct and validate DAML+OIL XML files for the ontologies [8, 178]. OilEd is a visual editor for ontologies, which calls the FaCT server to validate the ontology, and generates DAML+OIL XML files.

4.1. The Ontology for Person, Place, and Thing (PPT)

This section presents a detailed description of the implementation of the ontology for the upper level classes of PPT model.

4.1.1. The PPT Ontology

Following the PPT model defined in section 2, the PPT ontology is defined as six classes: **Person**, **Place**, **Thing**, **PersonDescription**, **PlaceDescription**, and **ThingDescription**. (Figure 52) Each of the three top-level classes are disjoint (i.e., **Person** is not a **Place**, etc.), each class must have at least one description class, and each description must describe exactly one object.

In this design, the description of the object is a disjoint class from the object described. The description classes represent the metadata as first class objects in the model. The description could be considered attributes of the object itself rather than a separate object, which would be much simpler.

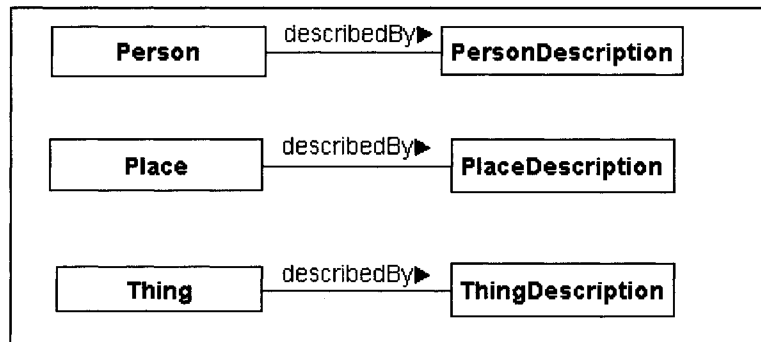


Figure 52. The Top-Level Classes

Defining the description as a separate class has several advantages:

1. A given class of object potentially might have more than one description, e.g., from different application views.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2. It is useful to be able to create new views of object, i.e., to create new attributes or filtered combinations of attributes, without changing the definition of the object itself.
3. Many services, such as registries and traders deal in descriptions of objects, rather than objects, e.g., they will deal with a **ThingDescription**, rather than a **Thing.**

However, the separation of an entity from its description is not essential for the ontologies or prototype implementation. For most purposes, the description classes are the important part of the ontology.

Table 21 lists the six classes and their properties. The "describedBy" relation of is represented by properties, e.g., the **Thing** class has a property called *descriptionOfThing*, with the range **ThingDescription**. In turn, the **ThingDescription** class has a property called describesPerson, with the range **Thing**. In this example, an instance of **Thing** is constrained to have one or more descriptions (i.e., instances of **ThingDescription**), and each instance of **ThingDescription** is constrained to be a description of exactly one instance of **Thing**. The **Person** and **Place** classes are defined similarly.

Table 21. The classes and properties of the top level ontology (after Figure 52).

Class	Properties:Range	Rules
Person	descriptionOfPerson:PersonDescription	Min 1
Place	descriptionOfPlace:PlaceDescription	Min 1
Thing	descriptionOfThing:ThingDescription	Min 1
PersonDescription	describesPerson:Person	Exactly 1
PlaceDescription	describesPlace:Place	Exactly 1
ThingDescription	describesThing:Thing	Exactly 1

4.1.2. The DAML Encoding, PPT.daml

These classes, properties, and relations are encoded in DAML+OIL XML (DAML). This section presents some details of the DAML encoding. Figure 53, Figure 54, and Figure 55 show fragments of a DAML file that represents the schema defined in Table 21.

The DAML file uses the Resource Description Framework (RDF) XML language [240]. The preamble states the XML namespaces (schemas) that are used (Figure 53, lines 2-6). This example shows the default namespaces used by any DAML file, including XMLSchema, RDF, RDF Schema, and, of course, DAML+OIL. The Dublin Core Element Set standard is used for the documentation (title, date, etc.) [228]. Other DAML or RDF Schemas could be imported as

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

namespaces, as will be shown in later sections. Following the XML standard, each namespace is given a local identifier, and the URL from which to retrieve the schema. For example, in Figure 53 line 2 declares the DAML+OIL schema to have the local name "daml", and to be imported from *http://www.daml.org/2001/03/daml+oil#.*

The first tag is <daml:Ontology>. This notation indicates that "Ontology" is defined in the XML namespace 'daml', which is defined in the namespaces as explained above. The Ontology object holds metadata about the DAML file (Figure 53, lines 7-15). This documentation is largely omitted to save space.

Figure 54 shows a fragment of the DAML XML that defines the **Thing** and **ThingDescription** and their properties, as defined in Table 21. The classes are defined by <daml:Class> tags, and properties are defined with <daml:ObjectProperty> tags. These tags use the RDF notation "rdf:about" to declare the class or property that is described. For example, the definition of the **Thing** class begins with the XML tag (Figure 54, line 30):

<daml:Class rdf:about="http://somewhere.net/PPT.daml#Thing">

where "http://somewhere.net/PPT.daml" is the URL of the DAML file containing this ontology.

The DAML fragment in Figure 54 shows the definition of four of the objects from Table 21: two classes **ThingDescription** (lines 17-28), **Thing** (lines 30-39), and two properties *descriptionOfThing* (lines 41-49), *describesThing* (lines 51-59). These definitions reference each other.

The **ThingDescription** is defined in Figure 54, lines 17-28. After the opening tag, two metadata fields are included <rdfs:label> and <rdfs:data> (lines 18-19). These tags are defined by the RDF standard [243, 244]. Other metadata may optionally be included, including tags from other schemas. This metadata can be included in any DAML definition, but will be omitted from the other examples here to save space.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <rdf:RDF xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
3     xmlns:dc="http://purl.org/dc/elements/1.1/"
4     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
5     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
6     xmlns:xsd="http://www.w3.org/2000/10/XMLSchema#">
7  <daml:Ontology rdf:about="">
8      <dc:title>&quot;The top level ontology: a very simple model of
9          objects and descriptions&quot;</dc:title>
10      <dc:date></dc:date>
11      <dc:creator></dc:creator>
12      <dc:description></dc:description>
13      <dc:subject></dc:subject>
14      <daml:versionInfo></daml:versionInfo>
15  </daml:Ontology>
```

Figure 53. Fragment of DAML for Thing and ThingDescription, as in Table 21. (1 of 3)

```
17  <daml:Class rdf:about="http://somewhere.net/PPT.daml#ThingDescription">
18      <rdfs:label>ThingDescription</rdfs:label>
19      <rdfs:comment><![CDATA[]]></rdfs:comment>
20      <rdfs:subClassOf>
21        <daml:Restriction daml:cardinalityQ="1">
22         <daml:onProperty rdf:resource="http://somewhere.net/PPT.daml#describesThing"/>
23              <daml:hasClassQ>
24                  <daml:Class rdf:about="http://somewhere.net/PPT.daml#Thing"/>
25              </daml:hasClassQ>
26          </daml:Restriction>
27      </rdfs:subClassOf>
28  </daml:Class>
29
30  <daml:Class rdf:about="http://somewhere.net/PPT.daml#Thing">
31    <rdfs:subClassOf>
32     <daml:Restriction daml:minCardinalityQ="1">
33     <daml:onProperty rdf:resource="http://somewhere.net/PPT.daml#descriptionOfThing"/>
34         <daml:hasClassQ>
35              <daml:Class rdf:about="http://somewhere.net/PPT.daml#ThingDescription"/>
36         </daml:hasClassQ>
37      </daml:Restriction>
38      </rdfs:subClassOf>
39  </daml:Class>
40
41  <daml:ObjectProperty rdf:about="http://somewhere.net/PPT.daml#descriptionOfThing">
42      <daml:inverseOf rdf:resource="http://somewhere.net/PPT.daml#describesThing"/>
43      <rdfs:domain>
44          <daml:Class rdf:about="http://somewhere.net/PPT.daml#Thing"/>
45      </rdfs:domain>
46      <rdfs:range>
47          <daml:Class rdf:about="http://somewhere.net/PPT.daml#ThingDescription"/>
48      </rdfs:range>
49  </daml:ObjectProperty>
50
51  <daml:ObjectProperty rdf:about="http://somewhere.net/PPT.daml#describesThing">
52      <daml:inverseOf rdf:resource="http://somewhere.net/PPT.daml#descriptionOfThing"/>
53      <rdfs:domain>
54          <daml:Class rdf:about="http://somewhere.net/PPT.daml#ThingDescription"/>
55      </rdfs:domain>
56      <rdfs:range>
57          <daml:Class rdf:about="http://somewhere.net/PPT.daml#Thing"/>
58      </rdfs:range>
59  </daml:ObjectProperty>
```

Figure 54. Fragment of DAML for Thing and ThingDescription, as in Table 21. (2 of 3)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
61  <daml:Class rdf:about="http://somewhere.net/PPT.daml#Person">
62      <daml:disjointWith>
63          <daml:Class rdf:about="http://somewhere.net/PPT.daml#Thing"/>
64      </daml:disjointWith>
65  </daml:Class>
66  <daml:Class rdf:about="http://somewhere.net/PPT.daml#Place">
67      <daml:disjointWith>
68          <daml:Class rdf:about="http://somewhere.net/PPT.daml#Thing"/>
69      </daml:disjointWith>
70  </daml:Class>
71  <daml:Class rdf:about="http://somewhere.net/PPT.daml#Thing">
72      <daml:disjointWith>
73          <daml:Class rdf:about="http://somewhere.net/PPT.daml#ThingDescription"/>
74      </daml:disjointWith>
75  </daml:Class>
76  <daml:Class rdf:about="http://somewhere.net/PPT.daml#PersonDescription">
77      <daml:disjointWith>
78          <daml:Class rdf:about="http://somewhere.net/PPT.daml#ThingDescription"/>
79      </daml:disjointWith>
80  </daml:Class>
81  <daml:Class rdf:about="http://somewhere.net/PPT.daml#PlaceDescription">
82      <daml:disjointWith>
83          <daml:Class rdf:about="http://somewhere.net/PPT.daml#ThingDescription"/>
84      </daml:disjointWith>
85  </daml:Class>
```

Figure 55. Fragment of DAML for Thing and ThingDescription, as in Table 21. (3 of 3)

The *descriptionOfThing* property is defined in Figure 54, lines 41-49. The <daml:ObjectProperty> tag begins the declaration, which includes the domain (lines 43-45) range (lines 46-48) and a declaration that the *describesThing* is the inverse of this property (line 42). Note that the domain and range are classes of the ontology, which are defined by the <rdf:about> tag, with the URL of the ontology and tag. For example, the domain is the class **Thing**, which is identified by the URL in line 44:

<rdf:about="http://somewhere.net/PPT.daml#Thing"/>.

The inverse property is similarly identified by the <rdf:resource> tag.

In this example, the properties of the classes are defined as DAML Properties with cardinality restrictions. This is expressed in DAML notation as an <rdfs:subClassOf> tag, with one or more <daml:Restriction> tags. For example, the **ThingDescription** has the *describesThing* property, which is limited to exactly one instance (daml:cardinalityQ="1"). This is defined in Figure 54, lines 20-27. Again, the property and class in this relation are specified by URLs.

The DAML definitions of **Place** and **Person,** and the other properties are analogous to the definitions related to **Thing**.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

One further detail can be added to the DAML file. It is intuitively obvious that the classes and properties defined in Figure 52 and Table 21 are meant to be mutually exclusive. An entity is either a **Thing** or a **Person**, but not both. And, of course, a **ThingDescription** is not a **Thing** and vice versa.

In the absence of distinguishing properties, an automated reasoning algorithm may deduce that these classes are indistinguishable, and therefore equivalent. To avoid this unintended inference, the assumptions can be explicitly asserted as axioms.

Figure 55 shows five DAML axioms that define a **Thing** is not a **Person** (lines 61-65), and **Thing** is not a **Place** (lines 68-70), a **Thing** is not a **ThingDescription** (lines 71-75) and so on. Additional axioms are needed declare the properties that are disjoint, and so.

The complete ontology for the Person, Place, and Thing concepts is encoded a DAML+OIL XML file, say 'PPT.daml'. The ontology can be proved to be logically consistent using description logic or other automated reasoning, as defined in Chapter 3.

The validated DAML ontology is made available used by other ontologies by placing it on the network, e.g., at the URL *http://somewhere.net/PPT.daml.* The other ontologies below will use the concepts and properties of this ontology by loading it from this URL.

4.2. Examples of Domain Ontologies

In this section, two example ontologies are developed to illustrate how domain ontologies are created. One ontology classifies the library resources: the books, articles, etc. of the library collection. This ontology is created from a standard, the OMG Bibliographic Query Service Specification [177]. The second ontology classifies devices that might be used in a library; printers, scanners, personal computers. This ontology is derived from the class hierarchy of Gaia system [206].

In each domain ontology, the ontology defines concepts and relations that are important for the domain. In order to integrate a domain ontology with the People, Places, and Things (PPT) model, the classes of the domain ontology should be defined to be specializations and extensions of the classes of the PPT ontology. This is usually done by identifying a few concepts that are synonyms or subclasses of the PPT.

4.2.1. Domain Ontology 1: Library Resources

The library ontology is a classification of the resources of a library, i.e., the collection of books, journals, etc. available to the library patrons. This ontology is an example of a domain-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

specific ontology, which should be developed by experts in the organization and classification of information collections (e.g., librarians).

In this case, several standards have been developed by the community of experts, notably the Dublin Core Element Set [228] and the Object Management Group (OMG) Bibliographic Query Service Specification [177] (which extends the Dublin Core). The Dublin Core has been defined in RDF (and therefore is compatible with DAML+OIL), so it can be directly used in this study. The OMG Bibliographic Reference specification is formally defined in BNF and CORBA IDL (see [177]), but not yet defined in XML.

This study implemented key parts of the OMG standard in DAML+OIL to illustrate the creation of an ontology from a standards document. However, the ontology developed here is drastically simplified. The ontology could well contain many other concepts. Most of attributes defined by the OMG specification are omitted to save space, as are full definitions of standard identifiers such as the International Standard Book Number (ISBN) [234], and so on. In addition to the classes and attributes, the ontology may also state axioms to define restrictions and relations.

4.2.1.1. The Library Ontology

Figure 56 shows the basic ontology for the resources of a library. Each **LibraryResource** is described by one **ResourceDescription**. The "describes" relation is implemented as properties on the **LibraryResource** (*DescribedBy*) and the **ResourceDescription** (*DescriptionOf*). This is analogous to the *describesThing* and *descirptionOfThing* relationships between **Thing** and **ThingDescription**. This ontology uses different terminology to illustrate the independent development of each domain ontology.

The resources of the library are classified following the OMG standard [177]: a **LibraryResource** is either a **Book**, a **Journal**, an **Article** (from a **Book** or **Journal**), a **Thesis**, a **Proceeding**, a **Technical Report**, a **Patent** or a **Web Resource** (Figure 57). Each class of **LibraryResource** has a corresponding class of **ResourceDescription**, organized as a parallel hierarchy of **BookDescription**, **ArticleDescription**, and so on (Figure 58). The two hierarchies are related by the *DescribedBy/DescriptionOf* properties.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

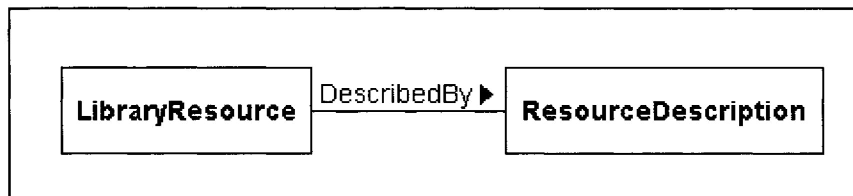


Figure 56. The base classes of the Library Ontology.

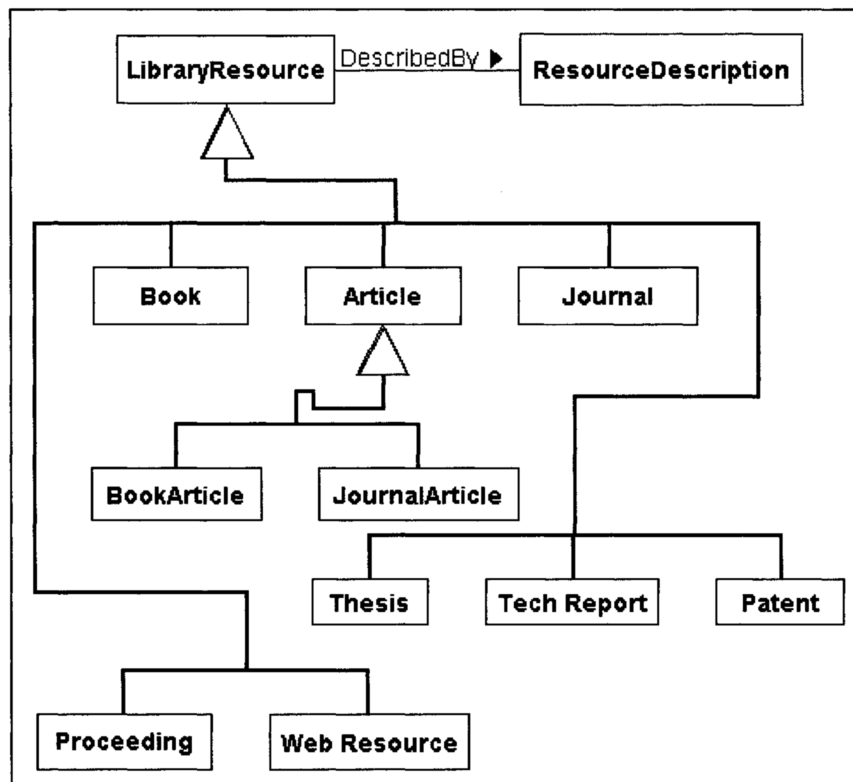


Figure 57. The classification of LibraryResources, based on [165].

Table 22 shows the classes and properties of the **LibraryResource** classes. In this trivial example, the objects have a single property, "DescribedBy", which refers to a **ResourceDescription**. Clearly, the objects might well have many other attributes, but this example focuses on the descriptions.

Table 23 shows the classes and properties of the **ResourceDescription** classes. The generic **ResourceDescription** imports the 'Dublin Core' ontology, which is the accepted standard for describing library resources. The Dublin Core provides a standard vocabulary for the most common descriptive terms, author ("creator"), title, publisher, etc. [228]. The OMG

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Bibliographic Reference standard extends the Dublin Core standard with additional attributes for each class of resource [177]. Table 23 defines the subset of the attributes from in [177] that are used in this study.

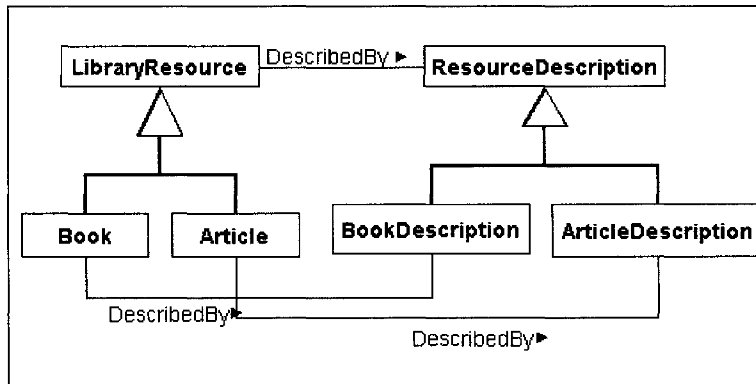


Figure 58. Some of the concepts for LibraryResource and ResourceDescription.

Table 22. The LibraryResource classes

Class	Attributes	Rules
LibraryResource	DescribedBy:ResourceDescription	Min 1
Book	DescribedBy:BookDescription	Min 1
Journal	DescribedBy:JournalDescription	Min 1
Patent	DescribedBy:PatentDescription	Min 1
Proceeding	DescribedBy:ProceedingDescription	Min 1
TechReport	DescribedBy:TechReportDescription	Min 1
Thesis	DescribedBy:ThesisDescription	Min 1
WebPage	DescribedBy:WebPageDescription	Min 1
Article	DescribedBy:ArticleDescription	Min 1
JournalArticle	DescribedBy:JournalArticleDescription	Min 1
BookArticle	DescribedBy:BookArticleDescription	Min 1

For example, the Dublin Core requires an identifier for each resource (the *<dces:identifier>* tag, row 2 of Table 23), while the OMG standard refines this to give each class a specific form of unique identifier. For a **Book**, the unique identifier is the ISBN (International Standard Book Number [234]), for an **Article** the unique identifier is the ISSN (International Standard Serial Number [122]), and for a **WebPage**, it is a URI (Universal Resource Identifier) [13].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 23. The ResourceDescription classes, adapted from [177]. Attributes with the prefix 'dces::' are defined in [228].

Class	Attributes	Rules
ResourceDescription	DescriptionOf Library:Resource	Min 1
	dces::identifier	
	dces::creator:Creator	Note: Creator ::= Person | Organization
	dces::contributor:Creator	
	dces::coverage	
	dces::date	
	dces::format	
	dces::language	
	dces::publisher	
	dces::relation	
	dces::rights	
	dces::source	
	dces::subject	
	dces::type	
	dces::title	
BookDescription	DescriptionOf:Book	Exactly 1
	ISBN_of:ISBN	See [234].
JournalDescription	DescriptionOf:Journal	Exactly 1
	ISSN_of:ISSN	See [122].
PatentDescription	DescriptionOf:Patent doc_number:string doc_office:string doc_type:string applicant:string	Exactly 1 Exactly 1 Exactly 1 Exactly 1 Exactly 1
ProceedingDescription	DescriptionOf:Proceeding	Exactly 1
TechReportDescription	DescriptionOf:TechReport	Exactly 1
ThesisDescription	DescriptionOf:Thesis	Exactly 1
WebPageDescription	DescriptionOf:WebPage	Exactly 1
	URL_of:URL	See [13].
ArticleDescription	DescriptionOf:Article	Exactly 1
	first_page:string	
	last_page:string	
JournalArticleDescription	DescriptionOf:JournalArticle	Exactly 1
	from_journal:string	Exactly 1
	volume:string	
	issue:int	
BookArticleDescription	DescriptionOf:BookArticle	Exactly 1
	from_book"string	Exactly 1

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The OMG standard defines dozens of attributes for the classes of resources. Table 23 includes a few of these attributes to give a flavor of how the class specific descriptions should be used. For example, a **PatentDescription** has a *doc_number* and other specific attributes. An **ArticleDescription** has a *first_page* and a *last_page*. A **BookArticleDescription** or a **JournalArticleDescription** has additional attributes to specify the source of the article.

To summarize, the ontology defines classes of objects, and a parallel set of classes of class-specific descriptions. The base description is the standard terms from the Dublin Core, and they are extended and specialized for the specific resources. E.g., the Dublin Core *<dces:identifier>* can be applied to any resource, but an <ISBN> is only valid for a **BookDescription** (and therefore, as a description of a **Book**), and so on.

4.2.1.2. The DAML Encoding

The ontology defined in Table 22 and Table 23 is encoded in DAML+OIL, similar to the PPT.daml, above. The DAML+OIL imports name spaces including the Dublin Core (*dces*) from *http://purl.org/dc/elements/1.1/*.

Figure 59 shows a fragment of a DAML+OIL file defining the **Book** class from Table 22. The **Book** class is defined in lines 1-15. The **Book** is a sub-class of **LibraryResource** (lines 2-4), and it has one property, *DescribedBy*, which must be a **BookDescription** and which must be a singleton (lines 5-14). The *DescribedBy* property is defined (lines 17-28), with its domain, range, in inverse. The other classes of resource are defined similarly.

Figure 60 shows DAML for the **BookDescription** and related classes and properties. The **BookDescription** is a sub-class of **ResourceDescription** (lines 2-5), and so inherits the properties of the superclass (dces:creator, dces:title, etc.) (not shown). Two other properties are defined for **BookDescription** (see Table 22), *DescriptionOf* (lines 28-40) and *ISBN_of* (lines 42-54). The range of the *ISBN_of* property is an instance of the class **ISBN** (lines 25-26). As in the previous examples, the **BookDescription** uses these properties with restrictions on the number of instances (lines 6-14 and 15-21).

The other classes of resource and resource description from Table 22 and Table 23 are defined similarly.

As explained in the previous section, the DAML needs to explicitly define precisely which classes are mutually exclusive. For example, while **Book** and **Article** are both instances of **LibraryResource**: a given entity must be either a **Book** or an **Article**, but not both. All the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

classes in Table 22 and Table 23 should be declared mutually disjoint with DAML axioms (not shown here).

The complete ontology for the library resources is encoded as a DAML+OIL XML file, say 'library.daml'. The ontology can be proved to be logically consistent using description logic. The validated DAML ontology is made available used by other ontologies by placing it on the Web, e.g., at the URL *http://somewhere.net/library.daml.* Other ontologies will use the concepts and properties of this ontology by loading it from this URL.

```
1 <daml:Class rdf:about="http://somewhere.net/library.daml#Book">
2   <rdfs:subClassOf>
3     <daml:Class rdf:about="http://somewhere.net/library.daml#LibraryResource"/>
4    </rdfs:subClassOf>
5    <rdfs:subClassOf>
6       <daml:Restriction daml:cardinalityQ="1">
7          <daml:onProperty
8             rdf:resource="http://somewhere.net/library.daml#DescribedBy"/>
9         <daml:hasClassQ>
10            <daml:Class
11              rdf:about="http://somewhere.net/library.daml#BookDescription"/>
12         </daml:hasClassQ>
13       </daml:Restriction>
14     </rdfs:subClassOf>
15 </daml:Class>
16
17<daml:ObjectProperty rdf:about="http://somewhere.net/library.daml#DescribedBy">
18      <daml:inverseOf
19         rdf:resource="http://somewhere.net/library.daml#DescriptionOf"/>
20      <rdfs:domain>
21         <daml:Class
22           rdf:about="http://somewhere.net/library.daml#LibraryResource"/>
23      </rdfs:domain>
24      <rdfs:range>
25        <daml:Class
26         rdf:about="http://somewhere.net/library.daml#ResourceDescription"/>
27      </rdfs:range>
28 </daml:ObjectProperty>
```

Figure 59. Fragment of DAML defining the Book.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
1 <daml:Class rdf:about="http://somewhere.net/library.daml#BookDescription">
2   <rdfs:subClassOf>
3    <daml:Class
4      rdf:about="http://somewhere.net/library.daml#ResourceDescription"/>
5   </rdfs:subClassOf>
6   <rdfs:subClassOf>
7    <daml:Restriction daml:cardinalityQ="1">
8     <daml:onProperty
9       rdf:resource="http://somewhere.net/library.daml#DescriptionOf"/>
10     <daml:hasClassQ>
11      <daml:Class rdf:about="http://somewhere.net/library.daml#Book"/>
12     </daml:hasClassQ>
13    </daml:Restriction>
14   </rdfs:subClassOf>
15   <rdfs:subClassOf>
16    <daml:Restriction daml:minCardinalityQ="1">
17  <daml:onProperty rdf:resource="http://somewhere.net/library.daml#ISBN_of"/>
18    <daml:hasClassQ>
19     <daml:Class rdf:about="http://somewhere.net/library.daml#ISBN"/>
20    </daml:hasClassQ>
21   </daml:Restriction>
22  </rdfs:subClassOf>
23 </daml:Class>
24
25 <daml:Class rdf:about="http://somewhere.net/library.daml#ISBN">
26 </daml:Class>
27
28 <daml:ObjectProperty
29        rdf:about="http://somewhere.net/library.daml#DescriptionOf">
30    <daml:inverseOf
31        rdf:resource="http://somewhere.net/library.daml#DescribedBy"/>
32    <rdfs:domain>
33       <daml:Class
34        rdf:about="http://somewhere.net/library.daml#ResourceDescription"/>
35    </rdfs:domain>
36    <rdfs:range>
37        <daml:Class
38        rdf:about="http://somewhere.net/library.daml#LibraryResource"/>
39    </rdfs:range>
40 </daml:ObjectProperty>
41
42 <daml:ObjectProperty
43       rdf:about="http://somewhere.net/library.daml#ISBN_of">
44      <rdfs:subPropertyOf
45       rdf:resource="http://purl.org/dc/elements/1.1/identifier"/>
46      <rdfs:domain>
47          <daml:Class
48       rdf:about="http://somewhere.net/library.daml#BookDescription"/>
49      </rdfs:domain>
50      <rdfs:range>
51          <daml:Class
52       rdf:about="http://somewhere.net/library.daml#ISBN"/>
53      </rdfs:range>
54 </daml:ObjectProperty>
```

Figure 60. Fragment of DAML for BookDescription.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.2.1.3. Relationship to the PPT Model

The ontology developed in this section is self-contained. It captures the concepts defined by the community of experts, without requiring knowledge of other domains.

In order to integrate this ontology with the PPT model, it is necessary to define relationships between the library ontology and the PPT ontology. In this example, the relationship is simple: **LibraryResource** is a subclass of **Thing**, and **ResourceDescription** is a subclass of **ThingDescription**. Figure 61 shows this fundamental linkage. From this simple definition, it is possible to deduce that all the subclasses of **LibraryResource** are subclasses of **Thing**, and therefore will be "cousins" of other subclasses of **Thing** from other ontologies such as the Device ontology defined below.

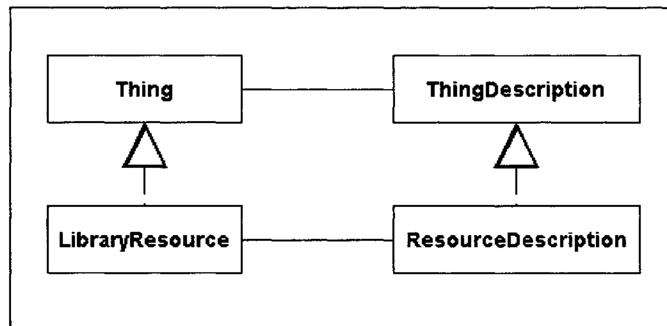


Figure 61. The fundamental linkage of the library resources and PPT.

```
 1 <?xml version="1.0" encoding="ISO-8859-1"?>
 2 <rdf:RDF xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
 3     xmlns:dc="http://purl.org/dc/elements/1.1/"
 4     xmlns:OilEd="http://img.cs.man.ac.uk/oil/OilEd#"
 5     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 6     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
 7     xmlns:xsd="http://www.w3.org/2000/10/XMLSchema#">
 8     <daml:Ontology rdf:about="">
 9     </daml:Ontology>
10      <daml:Class
11        rdf:about="http://somewhere.net/library.daml#LibraryResource">
12          <rdfs:subClassOf>
13              <daml:Class
14                rdf:about="http://somewhere.net/PPT.daml#Thing"/>
15          </rdfs:subClassOf>
16      </daml:Class>
17 </rdf:RDF>
```

Figure 62. A DAML+OIL XML file with an Axiom relating two ontologies.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

These relationships are defined in DAML, either in the Library ontology itself or in a separate XML document. In the latter case, the relationship is stated as a DAML Axiom, which refers to classes in two different ontologies. Figure 62 shows an example of an Axiom to declare that **library.daml#LibraryResource** is a subclass of **PPT.daml#Thing**.

4.2.2. Domain Ontology 2: Devices

A similar process is followed to create a second domain ontology, a classification for describing common digital devices. Ideally, this ontology would be developed by communities of experts, such as the IEEE and other standards setting organizations. At the time of the study, there was no generally accepted standard for classifying devices, so the ontology was based on the classification of devices in the Gaia system [206].

4.2.2.1. The Devices Ontology

Figure 63 shows a minimal ontology for **Device**, with each **Device** is described by a sub-class of **DeviceDescription**. Devices are categorized as **InputDevice** or **OutputDevice**. For the sake of this example, the ontology declares **InputDevice** and **OutputDevice** to be mutually exclusive, and each device category is mutually exclusive. This is not realistic, since real devices frequently have more than one function, including both input and output, as is indicated by the subclasses for **HardCopyIO**. For brevity, only a few types of device are shown.

Table 24 shows the attributes of **Device** and two of its sub-classes. As in the ontologies defined above, each device class has one attribute (*described_by*), which refers to its description. Clearly, the device classes might have other attributes, but this study concentrates on the descriptions.

Figure 64 and Table 25 show a part of the **DeviceDescription** ontology in more detail. All devices have an identifier, although for specific devices the identifier should use relevant standards. Additional attributes are defined which apply only to specific sub-classes, e.g., a **ScannerDescription** has attributes **Density** and **ErrorCorrection**, while **CameraProps** has an attribute for the number of **Colors**.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

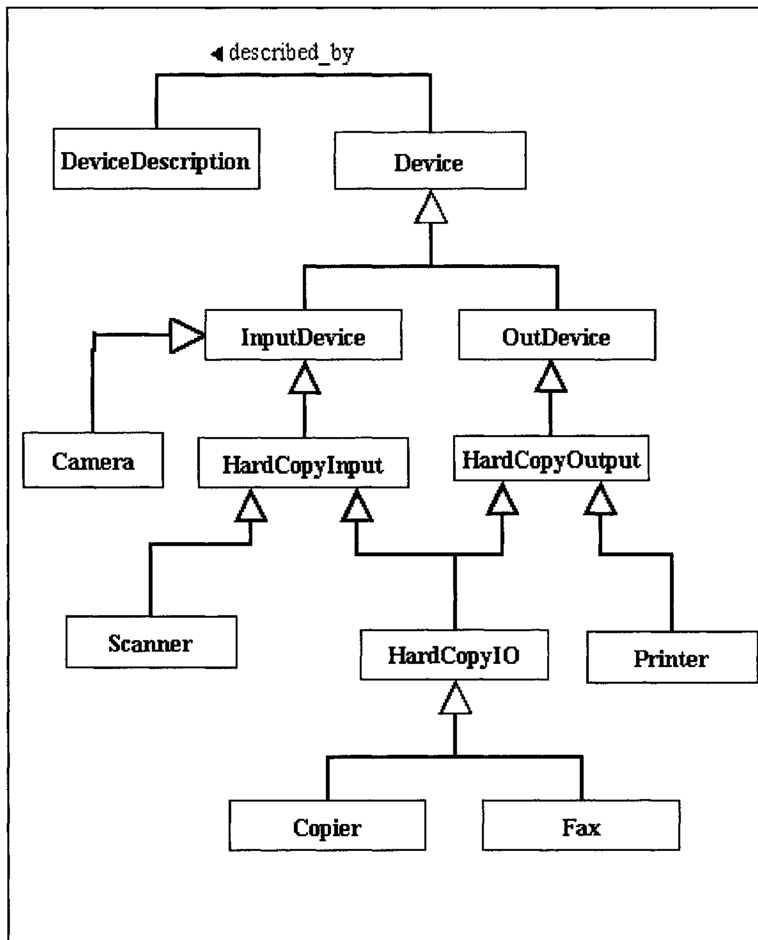


Figure 63. Part of the ontology for Devices.

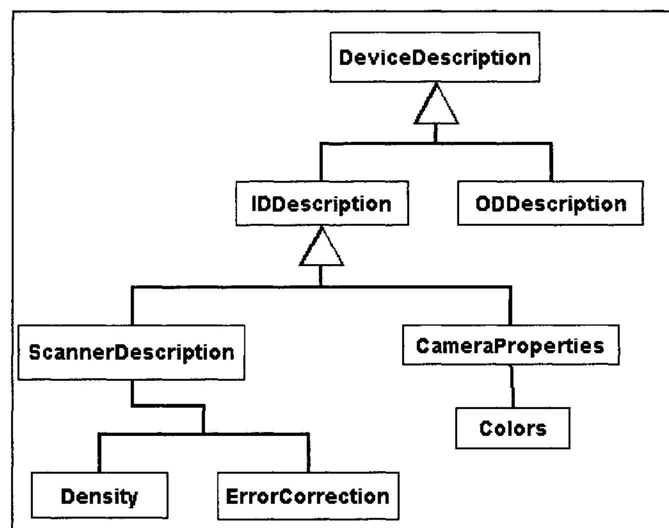


Figure 64. Part of the ontology for DeviceDescription.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 24. The attributes of the Device class hierarchy.

Class	Attributes	Rules
Device	described_by::DeviceDescription	Min 1
OutDevice	described_by::ODDescription	Min 1
InDevice	described_by::IDDescription	Min 1
Camera	described_by::CameraProps	Min 1
Scanner	described_by::ScannerDescription	Min 1

Table 25. Attributes of the DeviceDescription hierarchy.

Class	Attributes	Rules
DeviceDescription	description_of:Device	Exactly 1
	DeviceID:string	Exactly 1
	manual	
ODDescription	description_of:OutDevice	Exactly 1
IDDescription	description_of:InDevice	Exactly 1
CameraProps	description_of:Camera	Exactly 1
	Colors:int	
ScannerDescription	description_of:Scanner	Exactly 1
	Density:int	
	ErrorCorrection:boolean	

This ontology is similar to the library ontology described in the previous section, but it has some alternative terminology. For example, the *described_by* property here is equivalent to the *DescribedBy* property of the library ontology, and the description of a **Camera** is called **CameraProps**, rather than **CameraDescription**. These variations were deliberately included to simulate the natural result of autonomous ontology development: while an ontology can be made rigorous and internally consistent, it is not likely to align with all other terminologies.

4.2.2.2. The DAML Encoding

As in the previous examples, the ontology is encoded in a DAML XML file. The preamble with namespaces and the metadata will be similar to the examples above. The classes and properties are defined, as above.

Figure 65 shows a fragment of DAML that defines the **Device** (lines 1-12), **InputDevice** (lines 14-28) and **Camera** (lines 30-45), and the *described_by* property (lines 47-59). These classes are defined in Table 25. This example shows a hierarchy of classes: **Camera** is a sub-class of **InputDevice** (lines 31-33), which is a sub-class of **Device** (lines 15-17). Note that the *described_by* property is inherited, but is specialized in each class to refer to a corresponding

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

sub-class of **DeviceDescription**. For example, the **Camera** *described_by* property must be a **CameraProps** (which is a sub-class of **IDDescription**) (lines 35-44).

Figure 66 shows a fragment of DAML for the **DeviceDescription** and **IDDescription** (lines 32-47) classes as defined in Table 25. The **DeviceDescription** (lines 1-30) is the base description for all devices. The *description_of* property points to the device that is described. Two other properties are defined for all devices: *DeviceID* (lines 12-19) and the *manual* (lines 20-29). These properties are inherited by the sub-classes of **DeviceDescription**, such as **IDDescription** (lines 32-47).

Figure 67 shows the DAML for the description of a **Camera**, the **CameraProps**. The **CameraProps** is a sub-class of **IDDescription**, and has one property of its own, *Colors* (lines 64-71). Actually, the properties of devices are organized as a hierarchy (Figure 68). The *Colors* property (Figure 67, lines 74-85) is a sub-class of *camera_desc* (Figure 68, lines 117-125) which is a sub-class of *idev_desc* (Figure 68, lines 107-115), which is a sub-class of *device_description* (Figure 68, lines 101-105). All these properties are sub-properties of the *description_of* property (Figure 68, lines 87-99). This ontology illustrates an ontology with an elaborate terminology.

As in the case of the library ontology, the ontology should include rules to explicitly define which classes are disjoint, e.g., a **Camera** is not a **Scanner**, and so on. Again, for a real ontology, these rules will have to consider the possibility of a device with multiple functions, perhaps a device is both a scanner and camera.

The complete ontology for the devices is encoded as a DAML+OIL XML file, say 'device.daml'. The ontology can be proved to be logically consistent using description logic. The validated DAML ontology is made available used by other ontologies by placing it on the Web, e.g., at the URL, *http://somewhere.net/device.daml*. Other ontologies will use the concepts and properties of this ontology by loading it from this URL.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
1 <daml:Class rdf:about="http://somewhere.net/devices.daml#Device">
2   <rdfs:subClassOf>
3     <daml:Restriction daml:minCardinalityQ="1">
4         <daml:onProperty
5        rdf:resource="http://somewhere.net/devices.daml#described_by"/>
6         <daml:hasClassQ>
7           <daml:Class
8        rdf:about="http://somewhere.net/devices.daml#DeviceDescription"/>
9         </daml:hasClassQ>
10     </daml:Restriction>
11   </rdfs:subClassOf>
12 </daml:Class>
13
14 <daml:Class rdf:about="http://somewhere.net/devices.daml#InputDevice">
15   <rdfs:subClassOf>
16     <daml:Class rdf:about="http://somewhere.net/devices.daml#Device"/>
17   </rdfs:subClassOf>
18   <rdfs:subClassOf>
19     <daml:Restriction daml:cardinalityQ="1">
20         <daml:onProperty
21        rdf:resource="http://somewhere.net/devices.daml#described_by"/>
22         <daml:hasClassQ>
23             <daml:Class
24        rdf:about="http://somewhere.net/devices.daml#IDDescription"/>
25         </daml:hasClassQ>
26     </daml:Restriction>
27   </rdfs:subClassOf>
28 </daml:Class>
29
30 <daml:Class rdf:about="http://somewhere.net/devices.daml#Camera">
31   <rdfs:subClassOf>
32     <daml:Class
33        rdf:about="http://somewhere.net/devices.daml#InputDevice"/>
34   </rdfs:subClassOf>
35   <rdfs:subClassOf>
36     <daml:Restriction daml:cardinalityQ="1">
37         <daml:onProperty
38        rdf:resource="http://somewhere.net/devices.daml#described_by"/>
39         <daml:hasClassQ>
40             <daml:Class
41        rdf:about="http://somewhere.net/devices.daml#CameraProps"/>
42         </daml:hasClassQ>
43     </daml:Restriction>
44   </rdfs:subClassOf>
45 </daml:Class>
46
47 <daml:ObjectProperty
48      rdf:about="http://somewhere.net/devices.daml#described_by">
49   <daml:inverseOf
50        rdf:resource="http://somewhere.net/devices.daml#description_of"/>
51   <rdfs:domain>
52     <daml:Class
53        rdf:about="http://somewhere.net/devices.daml#Device"/>
54   </rdfs:domain>
55   <rdfs:range>
56     <daml:Class
57        rdf:about="http://somewhere.net/devices.daml#DeviceDescription"/>
58   </rdfs:range>
59 </daml:ObjectProperty>
```

Figure 65. Fragment of DAML for Camera.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
1 <daml:Class rdf:about="http://somewhere.net/devices.daml#DeviceDescription">
2     <rdfs:subClassOf>
3        <daml:Restriction daml:cardinalityQ="1">
4           <daml:onProperty
5       rdf:resource="http://somewhere.net/devices.daml#description_of"/>
6           <daml:hasClassQ>
7               <daml:Class
8       rdf:about="http://somewhere.net/devices.daml#Device"/>
9           </daml:hasClassQ>
10        </daml:Restriction>
11     </rdfs:subClassOf>
12     <rdfs:subClassOf>
13        <daml:Restriction daml:cardinalityQ="1">
14          <daml:onProperty
15       rdf:resource="http://somewhere.net/devices.daml#DeviceID"/>
16           <daml:hasClassQ
17       rdf:resource="http://www.w3.org/2000/10/XMLSchema#string"/>
18        </daml:Restriction>
19     </rdfs:subClassOf>
20     <rdfs:subClassOf>
21        <daml:Restriction>
22           <daml:onProperty
23       rdf:resource="http://somewhere.net/devices.daml#manual"/>
24           <daml:hasClass>
25               <daml:Class
26       rdf:about="http://somewhere.net/devices.daml#Manual"/>
27           </daml:hasClass>
28        </daml:Restriction>
29     </rdfs:subClassOf>
30 </daml:Class>
31
32 <daml:Class rdf:about="http://somewhere.net/devices.daml#IDDescription">
33     <rdfs:subClassOf>
34         <daml:Class
35      rdf:about="http://somewhere.net/devices.daml#DeviceDescription"/>
36     </rdfs:subClassOf>
37     <rdfs:subClassOf>
38        <daml:Restriction daml:cardinalityQ="1">
39           <daml:onProperty
40      rdf:resource="http://somewhere.net/devices.daml#description_of"/>
41           <daml:hasClassQ>
42               <daml:Class
43       rdf:about="http://somewhere.net/devices.daml#InputDevice"/>
44           </daml:hasClassQ>
45        </daml:Restriction>
46     </rdfs:subClassOf>
47 </daml:Class>
```

Figure 66. Fragment of DAML for DeviceDescriptions.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
49 <daml:Class rdf:about="http://somewhere.net/devices.daml#CameraProps">
50     <rdfs:subClassOf>
51         <daml:Class
52     rdf:about="http://somewhere.net/devices.daml#IDDescription"/>
53     </rdfs:subClassOf>
54     <rdfs:subClassOf>
55        <daml:Restriction daml:cardinalityQ="1">
56           <daml:onProperty
57     rdf:resource="http://somewhere.net/devices.daml#description_of"/>
58           <daml:hasClassQ>
59               <daml:Class
60      rdf:about="http://somewhere.net/devices.daml#Camera"/>
61           </daml:hasClassQ>
62        </daml:Restriction>
63     </rdfs:subClassOf>
64     <rdfs:subClassOf>
65        <daml:Restriction>
66           <daml:onProperty
67       rdf:resource="http://somewhere.net/devices.daml#Colors"/>
68           <daml:hasClass
69            rdf:resource="http://www.w3.org/2000/10/XMLSchema#integer"/>
70        </daml:Restriction>
71     </rdfs:subClassOf>
72 </daml:Class>
73
74 <daml:DatatypeProperty
75     rdf:about="http://somewhere.net/devices.daml#Colors">
76    <rdfs:subPropertyOf
77       rdf:resource="http://somewhere.net/devices.daml#camera_desc"/>
78    <rdfs:domain>
79         <daml:Class
80       rdf:about="http://somewhere.net/devices.daml#CameraProps"/>
81    </rdfs:domain>
82    <rdfs:range>
83         <xsd:integer/>
84    </rdfs:range>
85 </daml:DatatypeProperty>
```

Figure 67. Fragment of DAML for CameraProps description.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
87 <daml:ObjectProperty
88      rdf:about="http://somewhere.net/devices.daml#description_of">
89     <daml:inverseOf
90       rdf:resource="http://somewhere.net/devices.daml#described_by"/>
91     <rdfs:domain>
92         <daml:Class
93     rdf:about="http://somewhere.net/devices.daml#DeviceDescription"/>
94     </rdfs:domain>
95     <rdfs:range>
96         <daml:Class
97     rdf:about="http://somewhere.net/devices.daml#Device"/>
98     </rdfs:range>
99 </daml:ObjectProperty>
100
101 <daml:ObjectProperty
102    rdf:about="http://somewhere.net/devices.daml#device_desciption">
103     <rdfs:subPropertyOf
104       rdf:resource="http://somewhere.net/devices.daml#description_of"/>
105 </daml:ObjectProperty>
106
107 <daml:ObjectProperty
108    rdf:about="http://somewhere.net/devices.daml#idev_desc">
109    <rdfs:subPropertyOf
110       rdf:resource="http://somewhere.net/devices.daml#device_desciption"/>
111    <rdfs:domain>
112       <daml:Class
113         rdf:about="http://somewhere.net/devices.daml#IDDescription"/>
114    </rdfs:domain>
115 </daml:ObjectProperty>
116
117 <daml:ObjectProperty
118     rdf:about="http://somewhere.net/devices.daml#camera_desc">
119      <rdfs:subPropertyOf
120         rdf:resource="http://somewhere.net/devices.daml#idev_desc"/>
121      <rdfs:domain>
122          <daml:Class
123            rdf:about="http://somewhere.net/devices.daml#CameraProps"/>
124      </rdfs:domain>
125 </daml:ObjectProperty>
```

Figure 68. DAML for some of the device_description properties.

4.2.2.3. Relationship to the PPT Model

As in the case of the library ontology, the Device ontology is self contained. To integrate it with the PPT ontology, relationships between the Device ontology and the PPT ontology have to be defined. Again, this is simple: **Device** is a subclass of **Thing**, and **DeviceDescription** is a subclass of **ThingDescription**.

This definition assumes that the devices are not active objects, i.e., the "Printer" as an object, not a print service. This view is reasonable for some purposes, but might not be right for all applications. In that case, the Devices could be linked to other ontologies instead, such as the Gaia ontology below.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.3. Other Ontologies

The Semantic Web languages and tools can be used to create ontologies for many domains and applications. The following sections briefly describe two additional ontologies without a detailed commentary. Additional details are presented in Chapter 7, where the ontologies are used.

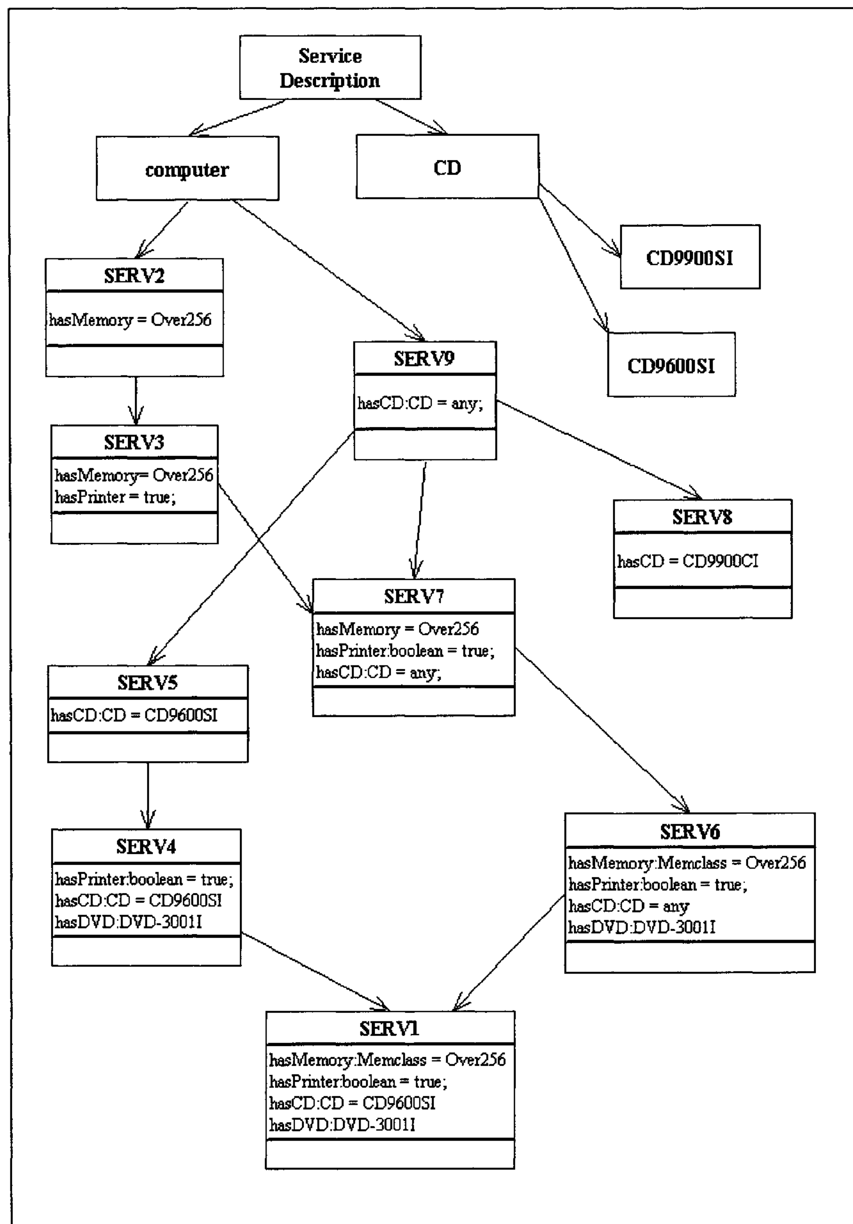


Figure 69. Hierarchy of classes, as in [80].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.3.1. Example from Gonzalez-Castillo et al.

Gonzalez-Castillo et al. present a class hierarchy for computing devices, which they use to define an algorithm for matchmaking ([80], page 8). Figure 69 shows a sketch of the class hierarchy defined in [80], with some details omitted. Note that the classes have multiple inheritance, they are not a simple tree. To test the implementation developed in subsequent chapters, this example hierarchy was defined as an ontology and encoded in DAML+OIL. The DAML is shown in Listing 1 in Appendix 1.

The mapping is natural and straightforward. The UML classes are defined as DAML classes, the UML attributes are properties with appropriate domains and ranges. Chapter 7 reports experiments using this ontology as a test case.

4.3.2. Gaia Applications

In related work, a case study examined the simple matchmaking scenario in which the user application queries to discover Gaia services [150]. Figure 70 shows a classification of some of the services available in a smart space managed by Gaia. This classification was defined as an ontology and encoded in DAML. The DAML is shown in Listing 2 in Appendix 1.

Gaia offers several services that can display information on different displays, e.g., from a PowerPoint file, from HTML or other text document, from JPEG or MPEG graphics, and so on. These services can be classified into related categories, as in Figure 70. The DAML ontology represents this classification. The DAML ontology can be loaded into the Ontology Server, and used to support queries and service composition [150].

The Gaia Context Server senses the environment of the smart room to detect important changes [199]. In other work, the concepts used in Gaia context were classified in an ontology [152]. Figure 71 shows some of the classes of context information. The DAML is shown in Listing 3 in Appendix 1.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

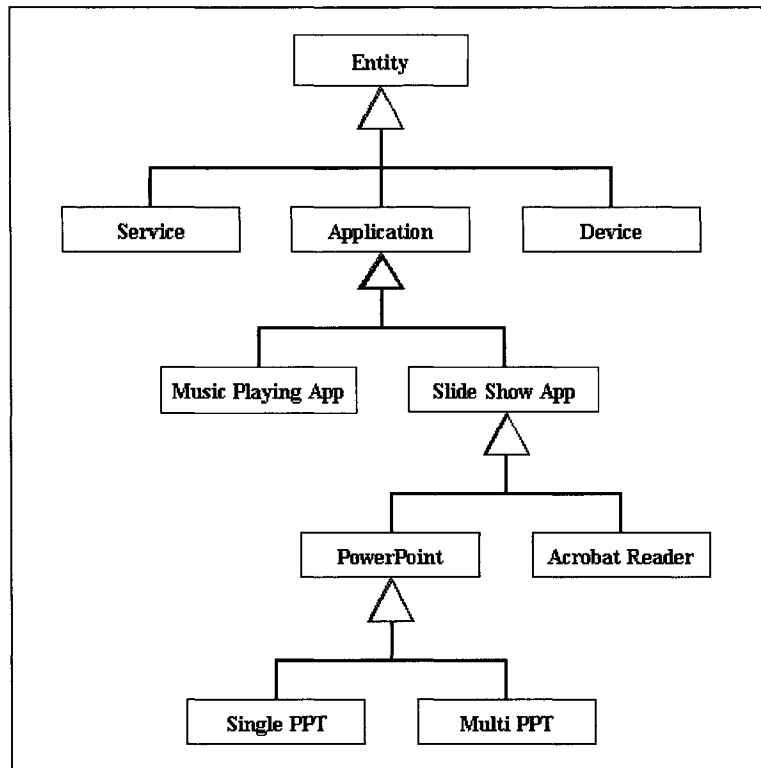


Figure 70. Part of the entity hierarchy of a smart space, with several categories of Application.

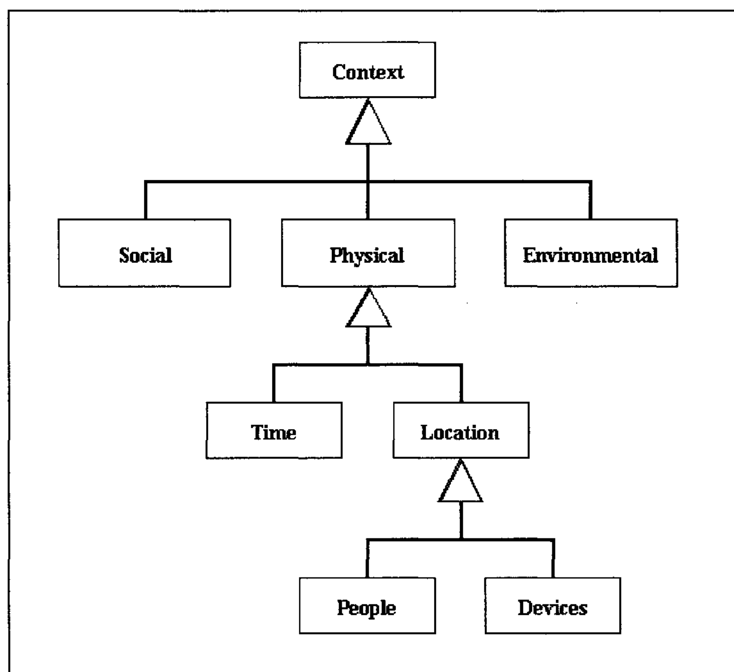


Figure 71. Classification of Context Information in Gaia (from [152]).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5. Summary

Chapter 3 presented a methodology for developing of ontologies based on the principle of decomposition: ontologies are developed for limited domains of interest, and then composed to form a larger overall ontology. This chapter demonstrated the methodology. Several example ontologies were developed, and then encoded in a standard XML language. These ontologies are the input for the prototype Ontology Server presented in the next Chapter.

The "Semantic Web Stack" defines an XML language to encode metadata, representing concepts of the abstract models. This study used the DAML+OIL (DARPA Agent Markup Language, Ontology Interchange Language) XML language, which has since been standardized as the Web Ontology Language (OWL). In addition to the universal portability of XML, this language has a formal semantics, which has been mapped to several formal logics. The formal semantics enables algorithmic manipulation and management of ontologies from many independent sources.

DAML+OIL (OWL) can express and reason about the vocabulary of the system, mostly the "nouns". This is necessary but not sufficient for a complete metadata language. For this reason, the Semantic Web stack almost certainly will grow, as the OWL language is used as a foundation for other languages and in some cases, it will be extended with additional concepts. Several of these languages are emerging, including, for example, RuleML [87] (for stating if-then-else rules), DAML-S/OWL-S [142] (for describing services), and RIE [124] (for stating policies).

At the apex of this hierarchy of ontologies are one or more *upper ontologies*. The upper ontology is a set of broad general concepts that apply to all domains. Two popular upper ontologies were used as the upper level of the ontologies developed in this chapter.

The Person, Place, and Thing concepts are the "top level" concepts for the ontologies in this study. The PPT concepts were mapped to both upper ontologies, demonstrating that the PPT is a valid ontology within those frameworks. The PPT concepts were developed for this application, and they have far more practical impact than the abstract upper ontologies.

The concepts in the ontologies need to be related to other ontologies of interest. For this study, the concepts in the library ontology should be correlated with the top-level ontology

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(PPT.daml) and the devices ontology defined in the next section. This requires discovery of terms and properties that are sub-classes or equivalent to concepts in other ontologies.

For example, the **LibraryResource** class is recognized to be a subclass of **PPT#Thing**. There may be other classes that relate to the PPT or to other ontologies. For example, the library ontology defines a concept **Author** (a subclass of **Creator**), which is a sub-class of **Person** in the PPT. The library ontology may also include additional concepts, such as Personal Roles for the library. These are subclasses of concepts in a high-level ontology Personal Roles (see [148, 149]).

Developing an ontology is labor intensive and time consuming. For instance, the Dublin Core standard (which is extremely simple) is the result of hundreds of person-hours by committed experts. The more elaborate OMG Bibliographic Reference Standard represents even more labor. The decomposition principle recognizes that significant effort must be expended to create domain ontologies, so the domains must be small enough to be tractable. The methods presented in this chapter do not eliminate the hard work, but they do make it easier to use, share, and extend the results.

This Semantic Web technology has been applied in a few Ubiquitous or Pervasive Computing research projects. Several groups are investigating the use of Semantic Web technology to support context-aware computing and Pervasive Computing (e.g., [25, 29, 38, 143, 173, 229, 230]). The ontologies described in this chapter are only a first step: significant effort will be required to define a comprehensive set of ontologies for Ubiquitous Computing. However, the prototype developed in the following chapters will work with any ontology that conforms to the Semantic Web standards.

Chapter 6 presents a prototype Ontology Server that implements the algorithms defined in Chapter 3. The prototype reads and parses the DAML+OIL XML ontologies described in this Chapter, to create and update a system ontology. The Ontology Service translates the XML into statements of Description Logic, which are asserted to a Knowledge Base. The Knowledge Base is used to prove facts about DAML+OIL XML files, including logical consistency and subsumption.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 6. Implementation: A Prototype Ontology Service

Chapter 3 defined a model and abstract architecture for semantic infrastructure for managing the metadata for a Ubiquitous Computing Environment. Chapter 4 presented a method for developing and encoding an ontology using DAML+OIL XML language. This chapter presents a prototype Ontology Service that implements the architecture and algorithms from Chapters 3 and 4.

The Ontology Service maintains a single, cumulative "current ontology" for a running system. The Ontology Service implements algorithms for validation and composition of ontologies; and for querying the Knowledge Base (KB) representing the facts implied by the composed ontologies.

The Ontology Service is built on top of a Knowledge Base (KB), which implements operations on DAML+OIL *ontologies*, using the "atomic" operations of the KB. The initial implementation used the CORBA FaCT server [10, 107, 108]. In subsequent work, a second Knowledge Base was added, based on the Racer logic engine [93, 95, 96].

1. Overview of the Prototype

This chapter presents a detailed account the interfaces and implementation of the prototype Ontology Service. Section 2 gives the interface of the Ontology Service. Section 3 presents the interface and implementation of the **OntoKB** class, with details of the use of the Knowledge Bases that are used as the foundation of the Ontology Service. Section 4 describes the implementation of the composition algorithm and section 5 presents the query algorithm.

Figure 72 shows a sketch of the Ontology Service in the system. The Ontology Service is a CORBA service that registers with the system infrastructure. The Ontology Service provides an interface used by applications and other services. The interface to the Ontology Service uses standard DAML+OIL XML. The Ontology Service interacts with a Knowledge Base (KB) through whatever interface is required. The KB is not directly accessible to users or applications.

Figure 73 shows the key components of the Ontology Service. The service has a CORBA interface, and two main components:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- The **Ontology Server**, which implements the interface, maintains the current ontology and other state information, and executes the algorithms defined in Chapters 3 and 4.
- The **OntoKB**, a private class that is a generic wrapper for the logic engine and KB.

The Ontology Service interface uses only public interfaces and formats, hiding the details of the data structures, logic engine, and KB. This makes it possible to substitute alternative implementations of the ontology data structures, logic engine, and KB.

The **OntoKB** class manages the details of interactions with the Knowledge Base. The initial implementation used the CORBA FaCT server [10, 107, 108]. The implementation also uses Java classes from the *uk.ac.man.cs.img.oil* package [8, 178]. These classes implement the access protocol for the CORBA FaCT server, parsing of DAML+OIL XML, and standard data structures for ontologies.

In subsequent work, a second Knowledge Base was added, based on the Racer logic engine [95, 96]. The alternative service was easy to add by implementing the **OntoKB** interface.

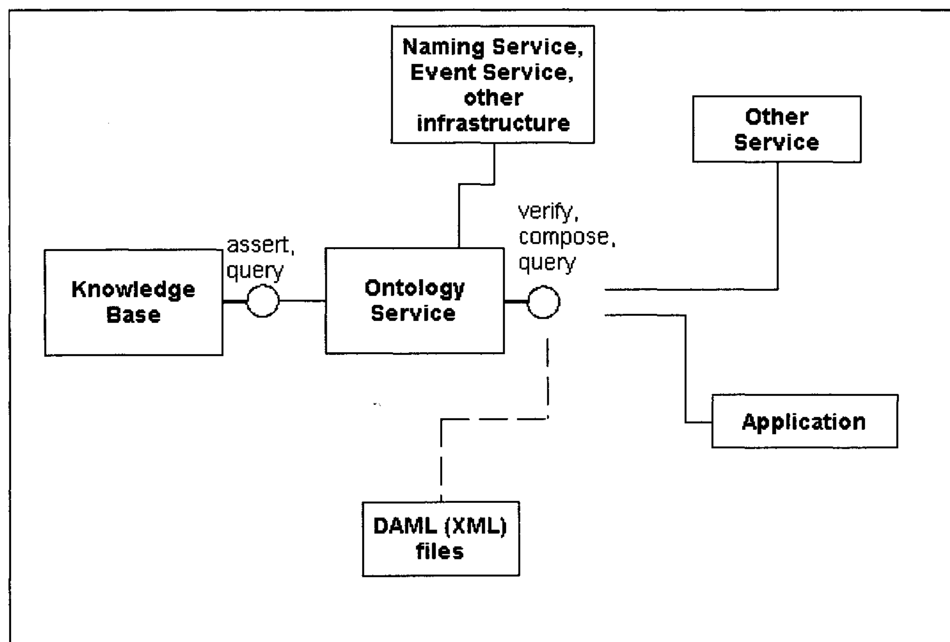


Figure 72. Sketch of the Ontology Service.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

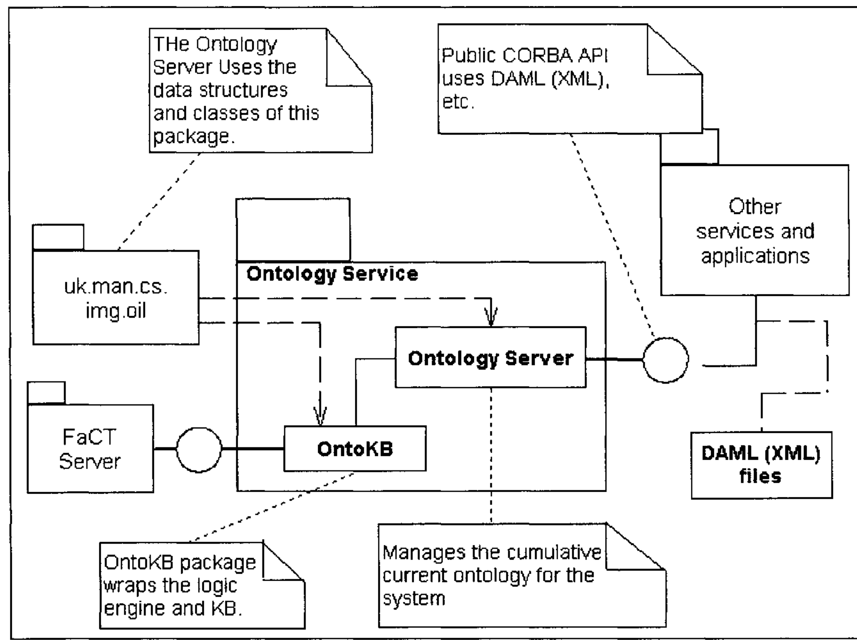


Figure 73. Overview of the Ontology Service.

2. Overview of the Ontology Server

2.1. The Current Ontology and Knowledge Base

The **Ontology Server** maintains a *current ontology*, which is loaded in the Knowledge Base and proved consistent (satisfiable). The current ontology is stored as an instance of the class *uk.ac.man.cs.img.oil.data.Ontology*. The *loadOntology* method updates the current ontology by adding the input ontology with the current ontology, to create the new current ontology.

The current ontology is represented by an instance of the Java class *uk.ac.man.cs.img.oil.data.Ontology*. This data structure is a taxonomy graph, which represents the classes and the subsumption relations (as well as properties and restrictions). Assuming that the ontology has been validated, as described in Chapter 3, the taxonomy graph represents the deductions of the Knowledge Base. For example, if class *A* is a child of *B* in the graph, then concept *B* subsumes concept *A* in the Knowledge Base. The Java data structure can be used as a "cache" for the Knowledge Base.

When the system is initialized, the current ontology is initialized to the empty ontology.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.2. The Ontology Service Interface

Figure 74 summarizes the interface of the Ontology Server, and Figure 75 shows a fragment of the CORBA IDL declaration. (Some debug methods are omitted for clarity.) The service has four operations, with several variations of queries.

The first operation, *clearAll* (Figure 75, lines 25), resets the current ontology. The second operation, *validateOntology* (Figure 75, lines 29-30) validates a DAML+OIL ontology. The third operation, *loadOnto* (Figure 75, lines 34-37) implements the composition operation defined in Chapter 4. The implementation of these operations are explained in Section 3 and 4 below.

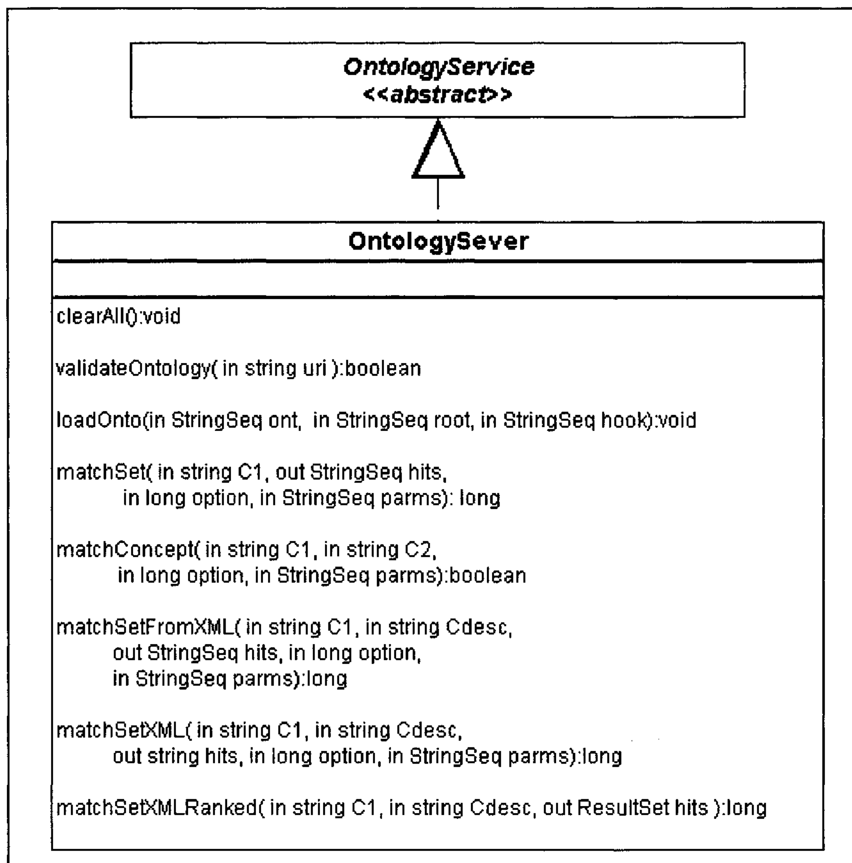


Figure 74. The OntologyServer class (implements the OntologyService interface)

The fourth group of operations are queries (Figure 75, lines 43-51). These interfaces implement similar queries, with variations on the input and output of the query. The query interface and implementation are defined in Section 5, below.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
1 module OntologyService
2 {
3    typedef sequence<string> StringSeq;
4
5    enum degree { NO, INTERSECT, SUBSUME, PLUGIN, EXACT };
6    struct ResultRec {
7      string Cname;
8      degree   degreeMatch;
9    };
10
11   typedef sequence<ResultRec> ResultSet;
12
13   exception InvalidOntology {
14     string reason;
15     };
16   exception OntologyNotFound {};
17   exception DuplicateOntology {};
18
19   //
20   // The Ontology Service interface
21   //
22   interface OntologyServer {
23
24     // clear the ontology and all tables
25     void clearAll();
26
27     // clear any previous ontology,
28     // load ontology from URL, then verify that it is consistent
29     boolean validateOntology( in string uri )
30       raises(OntologyNotFound);
31
32     //  load ontologies from DAML into existing ontology,
33     //  linking to existing classes (root,hook)
34     //  if result is not valid, raises InvalidOntology
35     void loadOnto(in StringSeq ont, in StringSeq root, in StringSeq hook)
36       raises (DuplicateOntology, OntologyNotFound, InvalidOntology
37       );
38
39     //
40     // experiments in semantic matching
41     // These interfaces are still evolving
42     //
43     boolean matchConcept( in string C1, in string C2);
44
45     long matchSet( in string Cname, out StringSeq hits, in long options );
46
47     long matchSetFromXML( in string Cname, in string Cdesc, out StringSeq hits, in long options);
48
49     long matchSetXML( in string Cname, in string Cdesc, out string hits, in long options);
50
51     long matchSetXMLRanked( in string Cname, in string Cdesc, out ResultSet hits );
52   };
53 };
```

Figure 75. CORBA IDL for OntologyService (some details omitted)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 26. Summary of the methods of OntologyServer. (See Section 3-5 for details.)

Operation	Description
void clearAll();	Resets the ontology and other state.
boolean validateOntology(in string uri) raises(OntologyNotFound);	Prove whether the ontology is valid. Input: *URI*: URL of DAML+OIL XML file Return: the result of the proof.
void loadOnto(in StringSeq ont, in StringSeq root, in StringSeq hook) raises (DuplicateOntology, OntologyNotFound, InvalidOntology);	Load one or more ontologies from DAML files. Input: *ont*[]: URL(s) of DAML+OIL Ontologies to be composed into the current ontology *root, hook:* "hints" to define how to connect input ontologies to other ontologies. Raises exception if fails.
boolean matchConcept(in string C1, in string C2);	Input: *C1*, *C2*: DAML class names Returns: *true* if *C1* matches *C2*
long matchSet(in string Cname, out StringSeq hits, in long options);	Input: *Cname*: the concept to match, a DAML class name *options*: 0 = match all super classes 1 = match only direct superclasses 2 = return list sorted Output: *hits*: List of concepts (DAML class names) Returns: Number of matches
long matchSetFromXML(in string C1, in string Cdesc, out StringSeq hits, in long options);	Input: *Cname*, *options*: same as above *Cdesc*: DAML+OIL XML to describe the concepts of the query Output: same as above Returns: Number of matches
long matchSetXML(in string Cname, in string Cdesc, out string hits, in long options);	Input: *Cname*, *Cdesc*: same as above Output: *hits*: A list of the concepts that match, formatted as a DAML+OIL XML document. Returns: Number of matches
long matchSetXMLRanked(in string Cname, in string Cdesc, out ResultSet hits);	Input: *Cname*, *Cdesc*: same as above Output: *hits*: A list of records, each record is a concept that matches, with the "degree of match". Returns: Number of matches

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 26 summarizes the methods of the interface. The interface uses standard data formats. Ontologies are defined in DAML+OIL and passed as URLs or XML text. Class names are defined using DAML+OIL, each class is defined within the XML namespace of its ontology. The results of the queries are returned in several forms, as a list of class names or as XML. The result set object is a weighted ranking of the results.

3. OntoKB: a Generic Interface to a Knowledge Base

The **OntoKB** class is a private class that provides a generic set of operations that use a Knowledge Base (KB) and logic engine. Figure 76 and shows the interface for this class. Table 27 describes the methods of the interface. (Miscellaneous test and debug operations have been omitted from this description.)

The **OntoKB** implements four groups of operations. The first operation (*loadOnto*) implements the composition algorithm. The second operation (*saveOntology*) writes the current ontology to a DAML+OIL XML file. The third operation (*validateOnt*) determines if a DAML+OIL ontology is valid.

The fourth group of operations define logical queries on the ontology. The *subsumes* query determines if the first class logically subsumes the second (i.e., the second class is a sub class of the first). The *compatible* query implements the compatibility test defined in Chapter 4. The interface provides alternative inputs to these queries.

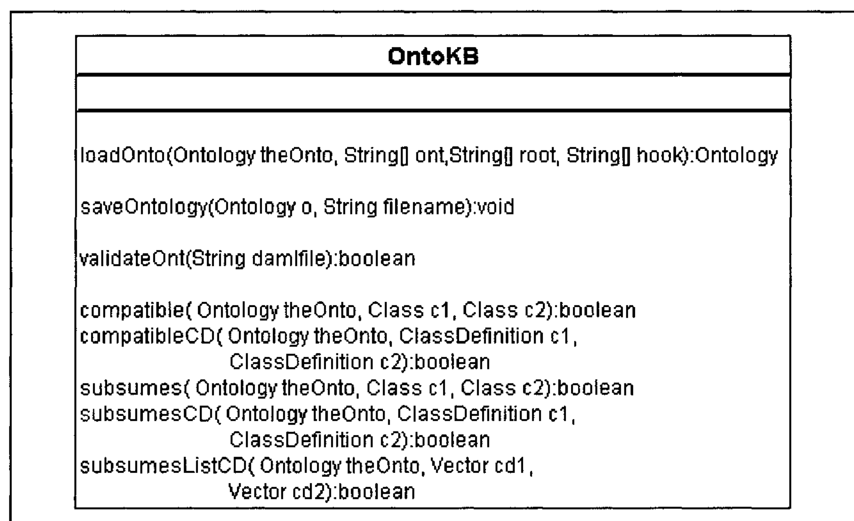


Figure 76. Summary of the OntoKB class.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 27. Definition of the Methods of the OntoKB.

Method	Description
Ontology loadOnto(Ontology theOnto, String[] ont, String[] root, String[] hook)	Compose one or more ontologies from XML files into a combined ontology Input: *theOnto:* the target ontology *ont[]*: ontologies to compose into *theOnto* *root*, *hook*: *links* between input ontologies and the target ontology. Returns: If valid, return the combined ontology.
void saveOntology(Ontology o, String filename)	Write Ontology DAML+OIL XML Input: *o*: Ontology in memory *filename*: file to write
boolean validateOntology(URL damlfile)	Check if ontology is logically consistent. Input: *damlfile*: URL of a DAML+OIL Ontology Returns: *true* if ontology is proved valid, *false* otherwise
boolean compatible(Ontology theOnto, Class c1, Class c2)	Implements the compatibility test for two classes of an ontology. Input: *theOnto*: the Ontology in memory *c1*, *c2*: classes to test Returns: *true* if *c1* is compatible with *c2* is proved valid, *false* otherwise
boolean comptibleCD(Ontology theOnto, ClassDefinition c1, ClassDefinition c2)	Same as above, but the input is the definition of *c1* and *c2*.
boolean subsumes(Ontology theOnto, Class c1, Class c2)	Determine if *subsumes(c1,c2).* Input: *theOnto*: the Ontology in memory *c1*, *c2*: classes to test Returns: *true* if *c1* subsumes *c2* is proved valid, *false* otherwise
boolean subsumesCD(Ontology theOnto, ClassDefinition c1, ClassDefinition c2)	Same as above, but the input is the definition of *c1* and *c2*.
boolean subsumesListCD(Ontology theOnto, Vector cd1, Vector cd2)	Same as above, but construct the conjunction of the class definitions in the two vectors.

The *compatible* and *compatibleCD* implement the same operation except using different classes from the *uk.ac.man.cs.img.oil.data* package. The *ClassDefinition* is a data structure with

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the full definition of the class, including all its super-classes and restrictions. The *ClassDefinition* can be an abstract combination of classes, it does not need to be a named class of the ontology. The *compatible* interface is a convenience interface used for simple cases.

The *subsumes* and *subsumesCD* methods are analogous to the *compatibility* interface described above. The *subsumesListCD* is a convenience function that tests subsumption between two classes formed by the conjunction of a list of classes. The first class is formed as the class:

(cd1[0] AND cd1[1] …),

where *cd1[0]* is the first element of the vector *cd1*, and so on.

In the prototype, the **OntoKB** implements a wrapper for the CORBA FaCT server [9, 10] or the Racer logic engine [95, 96]. The implementation used the Java package from OilEd, *uk.ac.man.cs.img.oil* [178]. This package includes classes to parse DAML+OIL XML files into Java data structures, to create a graph of classes. These data structure are used to construct the assertions and queries to the FaCT server or Racer.

3.1. Implementation of Automated Reasoning

This section briefly presents the Fast Classification of Terminology FaCT server, the package *uk.ac.man.cs.img.oil* (termed the Java OIL package below) [9, 10], and the Racer logic engine [95, 96].

3.1.1. The CORBA FaCT Server

The Fast Classification of Terminology (FaCT) server implements a Knowledge Base (KB) and automated reasoning using the SHIQ(D) subset of Description Logic[9, 10]. The CORBA FaCT server provides a CORBA interface around the Lisp reasoning engine. The CORBA IDL for the FaCT server is reproduced in Listing 4 in Appendix 1.

The FaCT server implements a simple interface, essentially *tell* (assert) and queries (e.g., *satisfiable* or *subsumes*). The input arguments are a string containing sequence of statements in the OIL language. This string can be encoded in several forms: the original text encoding for OIL, an XML encoding called “SHQ”, or another XML encoding called “DIG” [9, 10, 46]. The operation of the server is identical, the only difference is the format of the messages. This study used the older SHQ XML format [9, 10].

The FaCT server maps the statements of OIL language (e.g., encoded in SHQ XML) to Description Logic and implements a KB and automated reasoning. Description Logic and the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

reasoning algorithms were described in Chapter 4. The FaCT server is described in [108, 111, 112, 117].

The Knowledge Base is instantiated by encoding an entire DAML+OIL ontology as SHQ, and sending a series of assertions to the FaCT server. Every DAML+OIL statement is translated into the corresponding sequence of statements in SHQ XML, following the formal semantics of the DAML+OIL language [100]. The SHQ is passed to the CORBA FaCT server. Assuming the encoding was correct, proofs on the FaCT KB imply the consistency and subsumption of the DAML objects. For example, if all the OIL concepts (corresponding to the DAML classes) are *satisfied* in the FaCT KB, then the DAML ontology is logically consistent (contains no contradictions).

Queries to the KB may require concept as arguments, e.g., *satisfiable(concept).* This is implemented by constructing a fragment of SHQ that defines the concept to be tested. The query may be a class name, restrictions, relations, or logical combination of concepts.

3.1.2. The Java OIL Package

The Java OIL package *(uk.ac.man.cs.img.oil)* is a set of Java classes that wraps the CORBA FaCT server [8, 178]. The package provides Java classes to implement the communication protocols for the CORBA FaCT server. This Java package also provides a parser that reads DAML+OIL XML and creates Java data structures to represent the ontology, as well as methods to output the ontology as DAML+OIL, OWL, and other formats.

In the prototype Ontology Service, the Java OIL package is used to parse DAML+OIL XML (DAML) into Java data structures that represent DAML *Ontologies*, *Classes*, *Properties* and *Axioms*, and the relationships between the DAML objects. For example, the Java class *uk.ac.man.cs.img.oil.data.Ontology* has methods to discover all the DAML *Classes* in the Ontology and to look up a DAML *Class* by name (i.e., the URL for the class), and so on. The Java class *uk.ac.man.cs.img.oil.data.Class* represents a DAML Class, and has methods to retrieve its parents (super-classes), children (sub-classes), and properties.

The Java OIL package provides standard methods to "render" (write) the Java classes into SHQ or other intermediate language to send to FaCT. For example, the Java class

uk.ac.man.cs.img.oil.data.output.daml_oil_03_2001

has methods to traverse all the objects in an instance of *uk.ac.man.cs.img.oil.data.Ontology*, and write a DAML+OIL XML file to represent the whole ontology.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Finally, the Java OIL package also provides Java classes to connect and invoke methods on the CORBA FaCT server. The *img.fact* package provides the client connection to the Knowledge Base, and methods to load and query the KB.

This study uses the data structures of Java OIL package as the representation of the ontology throughout the Ontology Server. The algorithms described below operate on these data structures, e.g., to determine the siblings of a class, the methods of *uk.ac.man.cs.img.oil.data.Class* are called.

3.1.3. Summary and Example of the FaCT server

As discussed above, the Java OIL package and the FaCT server implement automated reasoning about DAML+OIL XML files through two mappings. The DAML+OIL XML is parsed into data structures and then mapped to OIL (SHQ or other encoding), then the OIL is mapped to Description Logic. Figure 77 illustrates this chain of translations.

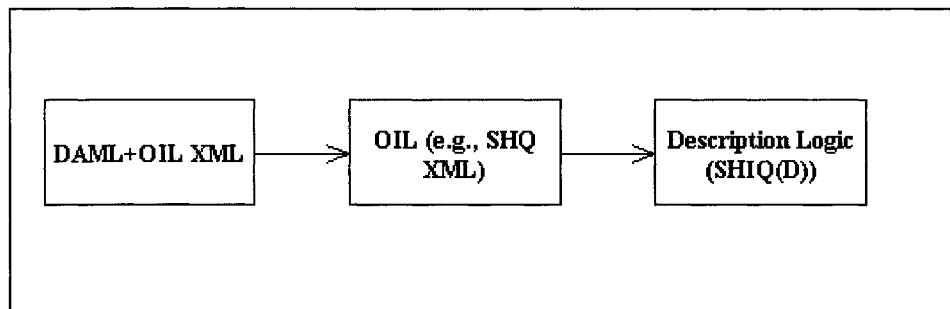


Figure 77. The logical mappings from DAML+OIL to Description Logic.

The Java OIL package implements all the operations required to load a DAML+OIL ontology from XML into the FaCT Knowledge Base. This operation is implemented in the following steps:

1. The DAML+OIL XML is parsed to create Java data structures in memory, an instance of *uk.ac.man.cs.img.oil.data.Ontology*. (This step is not needed if the ontology is already in memory.)
2. The ontology is written into intermediate code, e.g., the XML rendering of SHQ [9, 10]. This translation is implemented by the Java OIL package, e.g. *uk.ac.man.cs.img.oil.output.shiq*. (Different versions of FaCT have different, but equivalent, options for this step.)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3. The intermediate code is loaded to the FaCT server, to create a KB. This is implemented by classes of the Java OIL package that set up a connection to the CORBA FaCT server and invoke methods to execute assertions to the KB.

Assuming that these translations are correctly implemented, the proofs of the FaCT server can be applied to the Java data structures and the DAML+OIL XML. For example, if the KB is proved to be logically consistent, then we can infer that the Java OIL Ontology is valid, and the DAML+OIL XML is also valid. In short, *proofs using Description Logic on the Knowledge Base imply logical results for the data structures and the DAML+OIL XML.* This mapping is the foundation of the implementation of the Ontology Service.

To illustrate this process, consider the small example ontology shown in Figure 78. The ontology describes a **Printer**, which is a subclass of a **Device**. The **Printer** is defined to have one attribute, the size of paper it supports. The paper size is one of three possible **Forms**, **US standard**, **Legal**, and **A10**.

The ontology shown in Figure 78 can be written in DAML+OIL XML. Figure 79 shows an example DAML+OIL *Ontology*, with details omitted for space. Each class in the diagram is defined as a DAML *Class*, and the attribute *paper_size* is a DAML *ObjectProperty.*

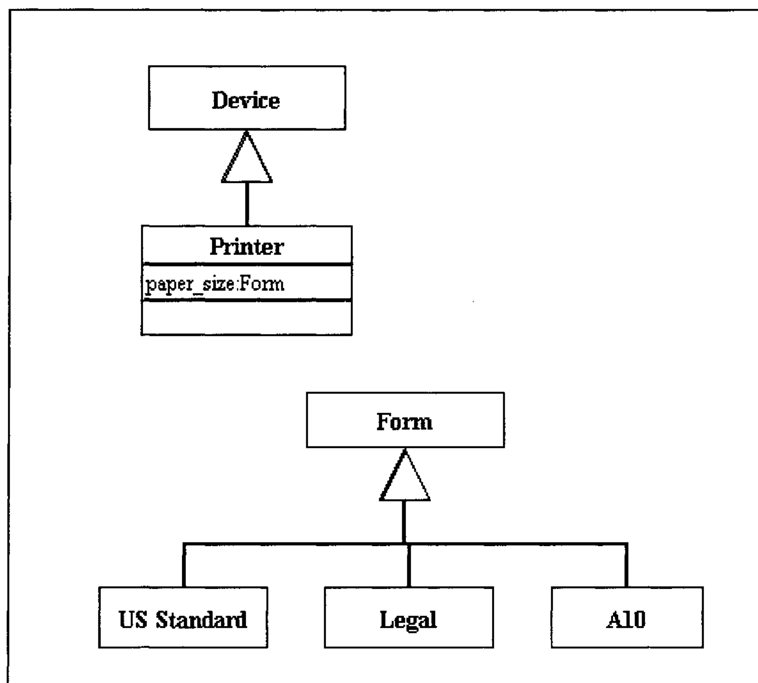


Figure 78. Some concepts from an example ontology.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
1
2 <daml:Class rdf:about="http://server.net/printer-ex.daml#Device">
3 </daml:Class>
4
5 <daml:Class rdf:about="http://server.net/printer-ex.daml#Printer">
6  <rdfs:subClassOf>
7   <daml:Class rdf:about="http://server.net/printer-ex.daml#Device"/>
8  </rdfs:subClassOf>
9  <rdfs:subClassOf>
10   <daml:Restriction>
11    <daml:onProperty rdf:resource="http://server.net/printer-ex.daml#paper_size"/>
12     <daml:hasClass>
13      <daml:Class rdf:about="http://server.net/printer-ex.daml#Form"/>
14     </daml:hasClass>
15   </daml:Restriction>
16  </rdfs:subClassOf>
17 </daml:Class>
18
19 <daml:ObjectProperty rdf:about="http://server.net/printer-ex.daml#paper_size">
20  <rdfs:range>
21   <daml:Class rdf:about="http://server.net/printer-ex.daml#Form"/>
22  </rdfs:range>
23 </daml:ObjectProperty>
24
25 <daml:Class rdf:about="http://server.net/printer-ex.daml#Form">
26 </daml:Class>
27
28 <daml:Class rdf:about="http://server.net/printer-ex.daml#USStandard">
29  <rdfs:subClassOf>
30   <daml:Class rdf:about="http://server.net/printer-ex.daml#Form"/>
31  </rdfs:subClassOf>
32 </daml:Class>
33
34 <daml:Class rdf:about="http://server.net/printer-ex.daml#A10">
35  <rdfs:subClassOf>
36   <daml:Class rdf:about="http://server.net/printer-ex.daml#Form"/>
37  </rdfs:subClassOf>
38 </daml:Class>
39
40 <daml:Class rdf:about="http://server.net/printer-ex.daml#Legal">
41  <rdfs:subClassOf>
42   <daml:Class rdf:about="http://server.net/printer-ex.daml#Form"/>
43  </rdfs:subClassOf>
44 </daml:Class>
```

Figure 79. Example of DAML+OIL corresponding to the diagram in Figure 78.

The definition of the DAML *Class* “Printer” includes two DAML *Restrictions*, *subclassOfDevice* (lines 6-8), and a restriction to require the *Property* “*paper_size*” (lines 9-16).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The latter "restriction" defines the *paper_size* relation to apply between a **Printer** and a **Form**. This is how DAML expresses the existence of attributes.

The DAML+OIL XML is parsed into data structures in memory. In this case, the data structure is an instance of the Java class *uk.ac.man.cs.img.oil.data.Ontology,* containing one instance of *uk.ac.man.cs.img.oil.data.Property* and five instances of *uk.ac.man.cs.img.oil.data.Class*. The sub-class and restrictions are defined for each *Class*. For example, the **Printer** would be represented by an instance of *uk.ac.man.cs.img.oil.data.Class*, with a link to its superclass, **Device**, and a definition of the Restriction, *hasClass paper_size.*

The ontology is validated by writing an intermediate code to send to the FaCT Server. In this case, the ontology is written in SHQ XML. Figure 80 shows the SHQ XML corresponding to the DAML+OIL objects in Figure 79.

The definition of the concept is one or more assertions that define it. In the SHQ, each atomic concept or role name is declared (e.g., <DEFCONCEPT>), and the restrictions are defined with an <IMPLIESC> ("implies concept") (or IMPLIESR, "implies role"). Each "implies" statement is an assertion to the Knowledge Base that the antecedent implies the consequent. For example, in Figure 80 the concept **Printer** is defined in lines 4-16. First, the name of the concept is defined (line 4). Then two assertions are stated (lines 5-16 state the two assertions together). The relationship "Printer subClassOf Device" is defined by the assertion, (Printer $\Rightarrow$ Device) (equivalently, Printer $\supseteq$ Device). The Restriction is defined by the assertion (Printer $\Rightarrow$ ($\exists$ paper_size Form)). Together, these assertions define the **Printer** concept.

As described in section 3.2 below, the ontology is loaded into the Knowledge Base by making a sequence of calls to the FaCT server, one for each assertion. Figure 81 shows a sketch of some Java code to load assertions defining the **Printer** concept into the KB. In this example, the connection to the CORBA FaCT Server is called 'factkb', and the assertions for the Printer are loaded by a series of method invocations. For each method invocation, the argument is extracted from the SHQ XML in Figure 80.

Once all the concepts are asserted to the KB, the KB is verified by testing that all the concepts are satisfiable, i.e., there is no logical contradiction. In the example, the KB is satisfiable if and only if all five concepts are satisfiable. This test is implemented by testing that each concept is satisfiable, with a series of queries to the KB.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
1
2 <DEFCONCEPT NAME="N0_DEVICE"></DEFCONCEPT>
3
4 <DEFCONCEPT NAME="N0_PRINTER"></DEFCONCEPT>
5 <IMPLIESC>
6  <CONCEPT><PRIMITIVE NAME="N0_PRINTER"></PRIMITIVE></CONCEPT>
7  <CONCEPT>
8   <AND>
9    <PRIMITIVE NAME="N0_DEVICE"></PRIMITIVE>
10    <SOME>
11     <PRIMROLE NAME="N0_PAPER_SIZE"></PRIMROLE>
12     <PRIMITIVE NAME="N0_FORM"></PRIMITIVE>
13    </SOME>
14   </AND>
15 </CONCEPT>
16 </IMPLIESC>
17
18 <DEFROLE NAME="N0_PAPER_SIZE"></DEFROLE>
19
20 <DEFCONCEPT NAME="N0_FORM"></DEFCONCEPT>
21
22 <DEFCONCEPT NAME="N0_USSTANDARD"></DEFCONCEPT>
23 <IMPLIESC>
24  <CONCEPT> <PRIMITIVE NAME="N0_USSTANDARD"></PRIMITIVE> </CONCEPT>
25  <CONCEPT>
26   <AND>
27    <PRIMITIVE NAME="N0_FORM"></PRIMITIVE>
28   </AND>
29  </CONCEPT>
30 </IMPLIESC>
31
32 <DEFCONCEPT NAME="N0_A10"></DEFCONCEPT>
33 <IMPLIESC>
34  <CONCEPT> <PRIMITIVE NAME="N0_A10"></PRIMITIVE> </CONCEPT>
35  <CONCEPT>
36   <AND>
37    <PRIMITIVE NAME="N0_FORM"></PRIMITIVE>
38   </AND>
39  </CONCEPT>
40 </IMPLIESC>
41
42 <DEFCONCEPT NAME="N0_LEGAL"></DEFCONCEPT>
43 <IMPLIESC>
44  <CONCEPT> <PRIMITIVE NAME="N0_LEGAL"></PRIMITIVE> </CONCEPT>
45  <CONCEPT>
46   <AND>
47    <PRIMITIVE NAME="N0_FORM"></PRIMITIVE>
48   </AND>
49  </CONCEPT>
50 </IMPLIESC>
```

Figure 80. The SHQ XML corresponding to the DAML+OIL in Figure 79.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
factkb.defconcept("N0_DEVICE"); // From Figure 80 line  2

factkb.defconcept("N0_PRINTER");  // From Figure 80 line 4

factkb.impliesc(            // From Figure 80 line 5-15
  "<CONCEPT><PRIMITIVE NAME=\"N0_PRINTER\"/></CONCEPT>",
  "<CONCEPT><AND> <PRIMITIVE NAME=\"N0_DEVICE\"/></CONCEPT>
    <SOME><PRIMROLE NAME=\"N0_PAPER_SIZE\"/>
           <PRIMITIVE NAME=\"N0_FORM\"/></PRIMITIVE>
    </SOME></AND></CONCEPT>");
```

Figure 81. Sketch of code to load lines 2 through 16 in Figure 80.

For example, the **Printer** concept is tested with a call to the factkb, something like:

```
factkb.satisfiable(
  "<CONCEPT><PRIMITIVE NAME=\"N0_PRINTER\"/><CONCEPT>");
```

The argument to this method the name of the concept (Printer), formatted in SHQ XML (N0_PRINTER).

The satisfiability test for a concept considers all the assertions in the KB. In the case of the **Printer** concept, the concept "<PRIMITIVE NAME=\"N0_PRINTER\"/>" is satisfiable only if the subclass relation and the *paper_size* relation create no contradiction.

In this simple example, there is no contraction, so every concept will be satisfiable and the Knowledge Base is satisfiable. From this proof, it is concluded that the DAML+OIL Ontology is valid.

3.1.4. Using the Racer Logic Engine

There are several alternative to the FaCT server, e.g. Pellet [214] (which uses a different Description Logic than used here), Java Theorem Prover (JTP) [60] (which uses forward and backward chaining), Ontoprise [181] (which uses F-Logic [128]), F-OWL [237] (which uses F-Logic). To demonstrate the use of the **OntoKB** interface, the Racer logic engine was added as an alternative to FaCT. Racer implements a subset of Description Logic, termed $ALCN_{R+}$, which is similar to the Description Logic used by FaCT without datatypes [93, 95, 96].

The implementation of the **OntoKB** operations is analogous to the FaCT server, except using the Racer interface. The Racer Server is access through the use of the Java package

de.uni.hamburg.informatik.jracer

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

which implements readers and writers for the racer intermediate language, along with a socket based client server protocol.

First, the ontology is asserted to the Knowledge Base. This is done by parsing the DAML+OIL XML into data structures using the Java OIL package. Then the ontology is written as a series of statements in intermediate language, using the Java OIL package, *uk.ac.man.img.oil.output.racer.* Figure 82 shows an example of the intermediate code, analogous to Figure 79.

The intermediate description is asserted to the Racer Knowledge Base by sending a series of formatted strings through a socket.

When all the statements are loaded, the coherence of the KB is tested through the query:

(check-tbox-coherence current-tbox)

The return value of this query indicates the result of the satisfiability test.

Similarly, the subsumption test is implemented by:

1. convert the DAML classes into Racer intermediate language
2. construct the Racer query:

   (concept-subsumes? (<<concept 1>>) (<,concept 2>>))

3. Send the query to the Racer server through a socket.

```
1 (in-knowledge-base /file.rcr x)
2
3 (define-primitive-concept N0_DEVICE *top*)
4
5 (define-primitive-role N0_PAPER_SIZE *top*)
6
7 (define-primitive-concept N0_FORM *top*)
8
9 (define-primitive-concept N0_PRINTER *top*)
10 (implies N0_PRINTER (and N0_DEVICE (some N0_PAPER_SIZE N0_FORM)))
11
12 (define-primitive-concept N0_USSTANDARD *top*)
13 (implies N0_USSTANDARD N0_FORM)
14
15 (define-primitive-concept N0_A10 *top*)
16 (implies N0_A10 N0_FORM)
17
18 (define-primitive-concept N0_LEGAL *top)
19 (implies N0_LEGAL N0_FORM)
20
21 (exit)
```

Figure 82. The Racer definitions, analogous to Figure 79.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.2. Implementation of the OntoKB

The **OntoKB** package implements operations that load and query the KB. The **OntoKB** interface hides the details of the implementations from the **Ontology Server**. The **OntoKB** uses the Java OIL package to implement operations using the FaCT CORBA server or Racer, as described above. This section states the key operations implemented by the **OntoKB** class.

3.2.1. Validate Ontology

An ontology is proved valid by mapping all the objects from the DAML+OIL XML (DAML) file to statements of Description Logic (or other logic). By definition, the ontology is valid if and only if all classes are satisfiable [8, 178].

This operation is implemented by the **OntoKB** using the KB and the Java OIL classes. The first steps are to parse the DAML and load the ontology into the KB, as described in section 3.1 above. The validation is done according to the design of the KB.

When using FaCT, the ontology is validated by a series of queries to the server.

1. Using the Java data structure *uk.ac.man.cs.img.oil.data.Ontology*, determine all the DAML classes in the ontology.
2. For each DAML class in the ontology, call the FaCT server *satisfies* query.
3. If all classes are satisfied, then the KB is consistent, the ontology data structure is valid, and the DAML+OIL XML is valid.

When using Racer, a single query to test validity of the ontology, *(check-tbox-coherence current-tbox).*

Thus, for an Ontology with *n* Classes, the validation requires *n* calls to the FaCT server or one call to Racer

3.2.2. Testing concept Subsumption

Given a valid ontology, *O*, the subsumption of one DAML+OIL class by another is defined in terms of the Description Logic representing the classes and the ontology.

Prerequisites: Ontology *O* is valid, and is loaded as a KB, as above.

Step 1: For DAML Classes *C1*, *C1* in *O*, construct two concepts that define *C1* and *C2.*

For FaCT, this is implemented by using the Ontology to construct the SHQ XML that defines each class. Using the Java data structures that represent ontology *O*, construct the definition of *C1, defClass1,* and *C2, defClass2,* as described in Section 2.1.

For Racer, the definitions *defClass1* and *defClass2* are written in the Racer language.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Step 2: Call the FaCT server to test the subsumption.

The concepts constructed above are used as the input argument to the FaCT method invocation: *subsumes(defClass1, defClass2)* or the Racer message *(concept-subsumes? (defClass1) (defClass2))*

Step 3: The KB tests the subsumption of the classes in the current KB and returns the result.

If the result is *true*, *C1* subsumes *C2* in the KB, and therefore the DAML class *Class1* subsumes *Class2*.

This test requires one call to the KB.

3.2.3. Checkpoint/Restore

The **OntoKB** implements a checkpoint function, *saveOntology*. The checkpoint function writes the current ontology into a DAML+OIL XML file, which can be reloaded to recreate the current Knowledge Base and Ontology. This operation can be used to save the valid ontology, provisionally modify it, and, if necessary, reload the original ontology.

3.2.4. The Compatibility Test

Given a valid ontology, *O*, the *compatibility* of two DAML+OIL classes is defined in terms of the Description Logic representing the classes and the ontology. This operation implements the algorithm defined in detail in Chapter 4.

3.2.4.1. The subsumption test

First, the parent of *C* and parent of *D* are tested for subsumption. For the Java OIL Class for *C* and *D*, look up the definition of the parent(s) of the classes.

Each parent of a class is a class expression, with three possible forms:

1. Empty (the class is the root of a tree). In this case, the parent is "TOP".
2. A primitive class.
3. A class expression.

The class expression may be a disjunction of class expressions or a conjunction of class expressions, and the class expression may include restrictions on properties.

Subsumption is tested by constructing two concepts, one from the description of the parents of *C*, and one the description of the parents of *D*, as described in Section 3.1, above. These two concepts are tested for subsumption. For the FaCT server, the concepts are constructed by rendering the expressions in SHQ using the *uk.ac.man.img.oil.output.shiq*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

package. For Racer server, the concepts are constructed by rendering the expressions using the *uk.ac.man.img.oil.output.racer* package.

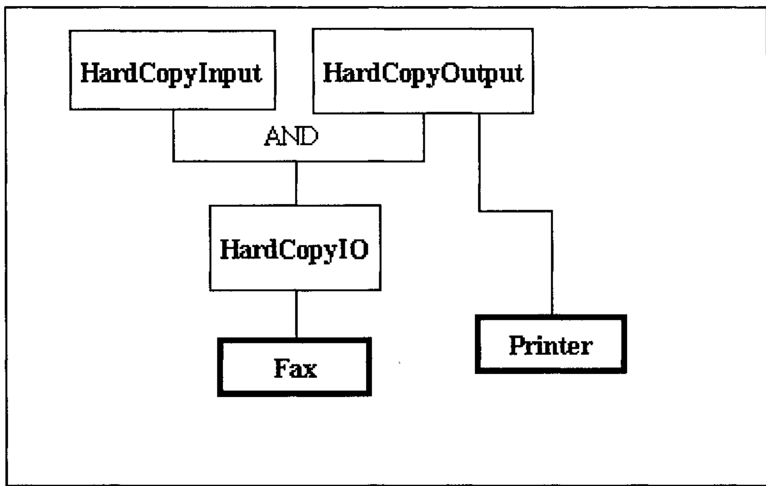


Figure 83. A simple ontology.

Figure 83 shows the example taxonomy of devices from Chapter 4. For FaCT, the test for compatibility of the parents of **Printer** and **Fax** would be would be something like Figure 84. For Racer, the test would be would be something like Figure 85. In either case, the test returns "true".

```
factkb.subsumes(
 "<CONCEPT><AND>
   <CONCEPT><PRIMITIVE NAME=\"N1_HARDCOPYINPUT\">
    </PRIMITIVE> </CONCEPT>
   <CONCEPT> <PRIMITIVE NAME=\"N1_HARDCOPYOUTPUT\">
  </PRIMITIVE></CONCEPT></AND></CONCEPT>",
"<CONCEPT> <PRIMITIVE NAME=\"N1_HARDCOPYOUTPUT\">
  </PRIMITIVE></CONCEPT>"
);
```

Figure 84. FaCT Server test for subsumes(parentOf(Printer), parentOf(Fax)).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
client. synchronizedSend("(concept-subsumes? ((and (define-primitive-concept
  hardcopyinput *top*) (define-primitive-concept hardcopyoutput *top*)) (define-primitive-
  concept hardcopyoutput *top*)))")
```

Figure 85. Racer Server test for subsumes(parentOf(Printer), parentOf(Fax)).

If the subsumption test returns *true*, then *C* and *D* are possible matches that must be tested for compatibility, as defined in Chapter 4.

3.2.4.2. Compatibility test

A query is formulated by defining a temporary concept that defines the conditions that must be met to indicate compatibility. The heuristic rules defined in Chapter 4 are implemented using the data structures that represent the DAML+OIL Ontology.

For two DAML classes *C* and *D*, the temporary class, *CandD*, is constructed, which has the following definition:

1. Create the class, (C AND D), i.e., define class CandD: ((definition of C) AND (definition of D))
2. Find the properties that C and D have in common
3. For each common property, assert a restriction that combines the restrictions of C and D.

The temporary class is tested for satisfiability, i.e., to determine if it creates a contradiction with the current ontology.

Table 28 lists some of the DAML+OIL properties and restrictions that correspond to the Description Logic restrictions used in the heuristics.

Table 28. Class constructors for Roles in Description Logic (see also [63, 80, 100, 102, 204]). (Concept C; Role R)

Description Logic Syntax	DAML+OIL XML	Restriction
$\forall R.C$	daml:toClass	Universal Role
$\exists R.C$	daml:hasClass	Existential Role
$\forall R.C$	daml:toClass	Universal Role
$\leq nR.C$	daml:minCardinalityQ	Cardinality
$\leq nR.C$	daml:maxCardinalityQ	"
$= nR.C$	daml:cardinalityQ	"
(datatype properties)		e.g., int, boolean, etc.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The heuristics defined in Chapter 4 are implemented using the DAML restrictions. Table 29 shows the combinations of restrictions with the definition of how they should be combined. Datatypes must match exactly (this is a match of the *type*, not the *value*). Any property of exactly the same type can be combined. The cardinality restrictions can be combined in an intuitive way. Table 30 shows the rules for the cardinalities.

Table 29. The combination rules for DAML+OIL properties.

	hasClass e_2	**toClass e_2**	**Cardinality**	**Datatype d_2**
hasClass e_1	hasClass $(e_1 \wedge e_2)$	toClass $(e_1 \wedge e_2)$	See Table 30	FAIL
toClass e_1		toClass $(e_1 \wedge e_2)$	See Table 30	FAIL
Cardinality			See Table 30	FAIL
Datatype d_1				If $(d_1 = d_2)$ then { datatype d_1 } else { FAIL }

Table 30. The combination rules for quantified properties.

	equalsQ m e_2	**maxQ m e_2**	**minQ m e_2**
hasClass e_1	equalsQ m $(e_1 \wedge e_2)$	maxQ m $(e_1 \wedge e_2)$	minQ m $(e_1 \wedge e_2)$
toClass e_1	equalsQ m $(e_1 \wedge e_2)$	maxQ m $(e_1 \wedge e_2)$	minQ m $(e_1 \wedge e_2)$
equalsQ n e_1	If $(n = m)$ { equalsQ n $(e_1 \wedge e_2)$ } else { FAIL }	If $(n \leq m)$ { equalsQ n $(e_1 \wedge e_2)$ } else { FAIL }	If $(n \geq m)$ { equalsQ n $(e_1 \wedge e_2)$ } else { FAIL }
maxQ n e_1		MaxQ (max (n, m) $(e_1 \wedge e_2)$	If $(n = m)$ { equalsQ n $(e_1 \wedge e_2)$ } else if $(n > m)$ { (minQ m e_2) $\wedge$ (maxQ n e_1) } else { FAIL }
minQ n e_1			minQ (min(m,n)) $(e_1 \wedge e_2)$

Continuing the example of devices, Figure 86 gives a sketch of definitions for Printer and Fax. The compatibility test constructs a query using the heuristics defined above. The concept to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

be tested would be something like the pseudocode in Figure 87. When this concept is tested, it is not satisfiable. The restriction on the *paperSize* property is not satisfiable: there is no concept in the KB for which (legal OR US standard) AND A10) is true.

```
Define Printer:
   (HardCopyOutput
   AND hasClass printDensity 600DPI
   AND hasClass paperSize (US standard OR legal)
   AND hasClass orientation (portrait OR landscape) )
Define Fax:
   (HardCopyIO
   AND  hasClass paperSize A10
  AND  hasClass baudRate 64KBS
  AND  hasClass orientation landscape )
```

Figure 86. Pseudocode defining the Printer and Fax concepts.

```
Define Printer AND Fax:
  ( (HardCopyOutput AND (HardCopyInput AND HardCopyOutput)
    AND hasClass printDensity 600DPI
   AND hasClass paperSize    ( (US standard OR legal) AND A10)
   AND hasClass orientation   ((portrait OR landscape) AND portrait)
   AND hasClass baudRate       64KBS

  Note: Assume that the ontology defines 'US standard', 'legal', and 'A10' to be disjoint
  (mutually exclusive), and 'portrait' and 'landscape' to be disjoint.
```

Figure 87. Pseudocode for the definition of (Printer and Fax) (from above).

3.2.4.3. Discussion

The *compatibility* test requires a variable number of calls to the KB, depending on the ontology and the query. In all cases, the subsumption test must be executed, which requires one call to the KB, plus the compatibility test requires one call to the KB per pair of concepts to be tested.

4. Implementation of the Composition Algorithm

The prototype **Ontology Server** implemented the composition algorithm defined in Section 6 of Chapter 3. The composition was implemented in a sequence of calls to the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Ontology Server. At each step, the input ontology or ontologies are composed with the current composite ontology. Axioms are input as a DAML+OIL ontology in the list of ontologies to be added. The implementation used the classes and data structures of the Java OIL package, and calls to the **OntoKB** to update the Knowledge Base and prove consistency, as presented in section 4.3, above.

This section explains the prototype implementation. Section 4.1 describes the Knowledge Base and the current system ontology. Section 4.2 defines the input to the composition. Section 4.3 describes the composition step, and section 4.4 explains how axioms were implemented. Section 4.5 explains the results of the composition, and section 4.6 gives an example.

Recall that the **Ontology Server** maintains a *current ontology*, which is loaded in the Knowledge Base and proved consistent (satisfiable). The current ontology is stored as an instance of the class *uk.ac.man.cs.img.oil.data.Ontology*. The *loadOntology* method updates the current ontology by adding the input ontology with the current ontology, to create the new current ontology.

When the system is initialized, the current ontology is initialized to the empty ontology. At the beginning of the composition algorithm, a snapshot of the current ontology is saved. The snapshot can be reloaded, to restore the original state if the composition fails.

4.1. Input to the Composition

The input to the composition is::

1. One or more URLs, which point to DAML+OIL XML files.
2. An optional list of links: pairs of DAML Class names, of the form *(root, hook).* For each pair, *root* is defined to be a sub class of *hook.*

4.2. Composing the Ontologies

For each URL representing an input ontology, the DAML+OIL file is downloaded, and parsed into an instance of the class *uk.ac.man.cs.img.oil.data.Ontology*.

The input *Ontology* is passed to the *OntoKB.loadOnto()* method. First, the input ontology is added to the current ontology, to form a new combined ontology.

1. The input Ontology is validated as described in Section 3.2.
2. The input ontology is merged with the current ontology using the *uk.ac.man.cs.img.oil.data.Ontology.include()* method, which adds all the objects of the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

input ontology to the current ontology. The result of this step is a new, combined ontology (in Java data structures).

3. Optionally, the combined ontology can be validated at this stage. (Even though the two ontologies are valid before the merge, it is possible that a contradiction could be created by this step.)

Second, any *links* are processed one by one. First, the *root* and *hook* class are looked up in the combined ontology. Assuming no error is detected, the *root Class* is defined to be a sub class of the *hook Class* in the combined ontology.

In addition, all the immediate sub classes of the *hook* Class are looked up, and axioms added to assert that the root is logically disjoint from each of its sibling. Essentially, the hint is implemented as defining *root* (and by implication, all its descendants) to be a new and distinct child of *hook*.

For example, Figure 88 shows a simple case where 'X' is to be linked as a sub class of 'A'. The link is added, to make 'X' a sibling of 'B' and 'C' (Figure 89). Then, to complete the intended meaning of the link, two axioms are added to assert the 'X' is disjoint from 'A' and 'B' (Figure 90).

This step was necessary to assure that the *root* Class is not subsumed (and confused with) other similar sub classes. This confusion is not unusual, because there is often insufficient information in the Knowledge Base to distinguish the high level concepts from different ontologies. In this case, the automated reasoning may deduce that some of the classes are equivalent, and coalesce them into a single concept. For example, Figure 91 shows an example where, in the absence of facts to the contrary, the concept 'X' is deduced to be the logically equivalent to the concept 'C'.

After the hints are added to the ontology, the combined ontology, now including the links, is validated, as described in section 3.2.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

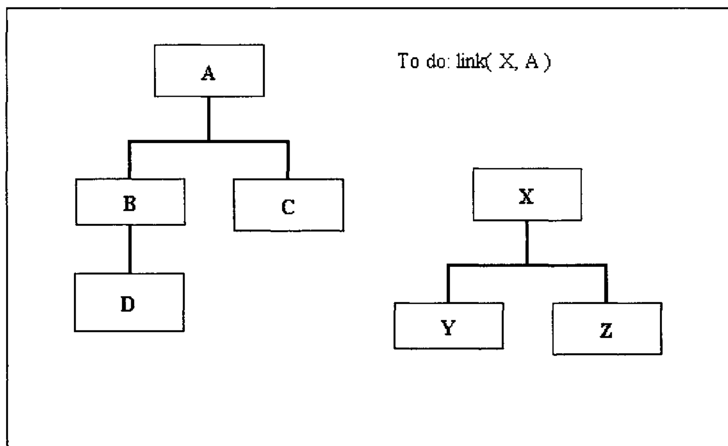


Figure 88. Merging two simple ontologies: X will be a subclass of A.

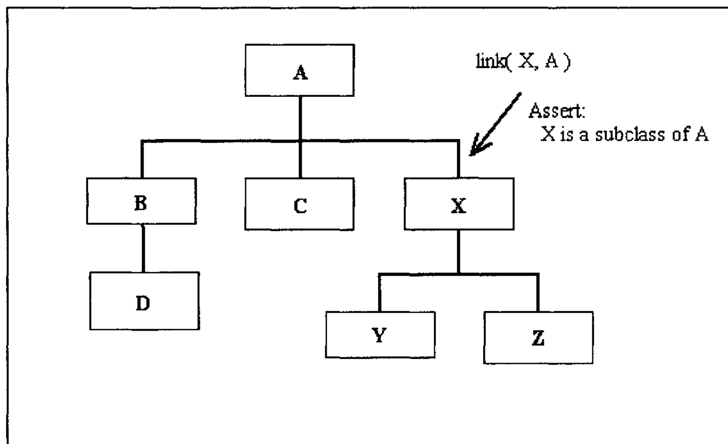


Figure 89. The link has been made: X, B, and C are siblings.

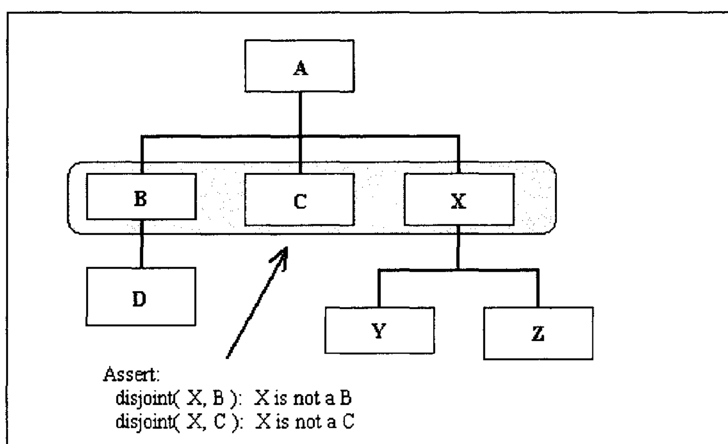


Figure 90. Assert that X is not an A or B.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

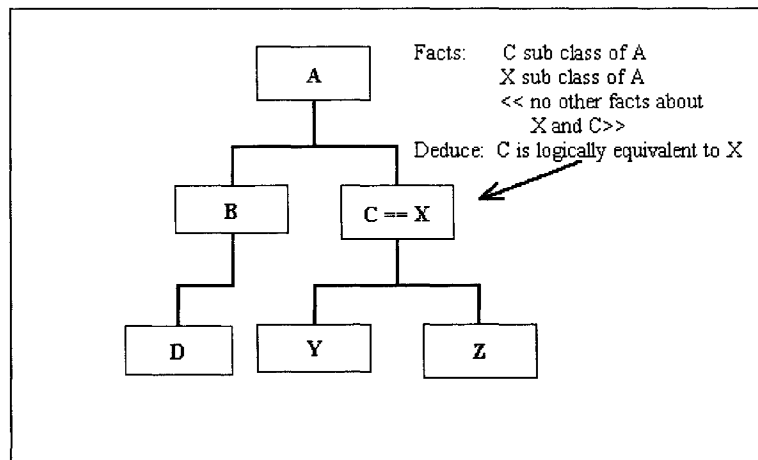


Figure 91. In the absence of sufficient information, two concepts may be deduced to be equivalent in the Knowledge Base.

4.3. Axioms (Optional Assertions)

In the composition algorithm defined in Chapter 4, an optional set of arbitrary assertions (axioms) can be included.

The axioms are implemented as one or more DAML+OIL XML files. These files do not need to contain whole ontologies, they need only the Axioms and any definitions needed by the Axioms. Note that these files can contain any Axioms desired, because they use the DAML XML name spaces to refer to ontologies.

The Axioms are added to the ontology using the composition operation described above. The *loadOntology* method is passed the URL(s) of the Axioms. The Axioms are merged and validated using the same method described above. The links are empty in this case.

4.4. Return the Results

If the composition fails (e.g., due to invalid or missing input), an exception is raised and the current ontology is not changed.

If the composition succeeds, the new combined ontology becomes the current ontology, and the previous system ontology is discarded.

4.5. Example: Composing the Thing Ontology

Ontologies are developed as separate ontologies intended to be composed into a combined ontology. This section shows how these ontologies are composed following the algorithm given in Chapter 4. In this example, a single system ontology is constructed by

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

composing three ontologies, People, Places, and Things (PPT), Devices, and Library ontologies. This process is implemented by a series of calls to the 'loadOnto', method of the **Ontology Server**.

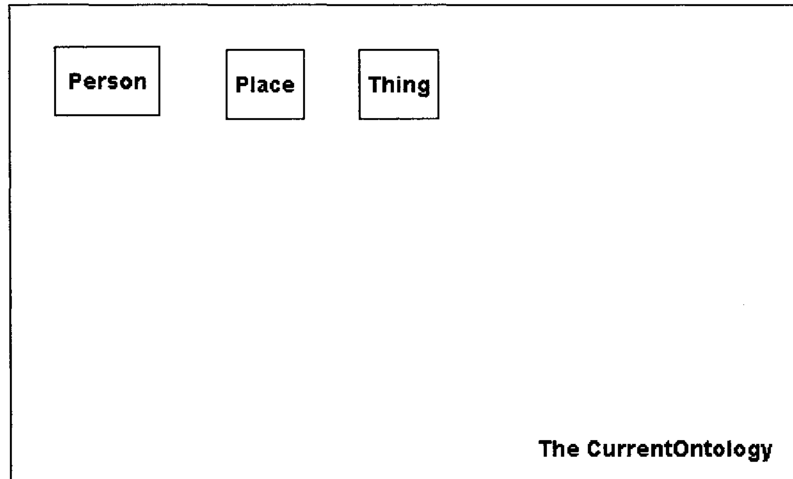


Figure 92. The initial ontology (details omitted for clarity).

Initially, the current ontology is the empty ontology. The **Person**, **Place**, and **Thing** (PPT) ontology is loaded by composing it with the empty ontology. Figure 92 illustrates part of the current system ontology at this step. Other ontologies can now be added, linking them to concepts in the PPT ontology as appropriate. Next, the Devices ontology is added to the system.

The Devices ontology defines a hierarchy of devices as a graph rooted in the concept **Device**, which has been defined to be a sub-class of **Thing**. The ontology also defines other concepts that are not part of the **Device** hierarchy, such as **On Line Manual** (i.e., on-line documentation) for a device. Each of the concepts in the ontology is identified by a URL, e.g., the concept **On Line Manual** is defined in the Library Ontology has the unique name, *http://somewhere.net/ontologies/devices.daml#OnLineManual.*

The input to the composition is the URL of the Devices ontology, and a link to define the relationship between **Thing** and **Device**. For example, the link would be defined with *hook[0]="Thing",* and *root[0]="Device".* This link will be implemented by adding the assertion *subClassOf("Thing", "Device")* to the ontology.

For example, Figure 93 shows a diagram of the system ontology composed of the PPT and the Devices ontologies. The composite ontology contains all the classes, relations, and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

axioms of the two input ontologies. In addition, in the new ontology the **Device** class and all its descendents are sub-classes (subsumed by) the **Thing** class. This ontology is validated, and is the new current ontology. The system ontology is represented by an instance of the Java class *uk.ac.man.cs.img.oil.data.Ontology*, with data structures to represent all the DAML Classes and Properties. The ontology is loaded into the Knowledge Base, and proved valid as described in section 2.2.

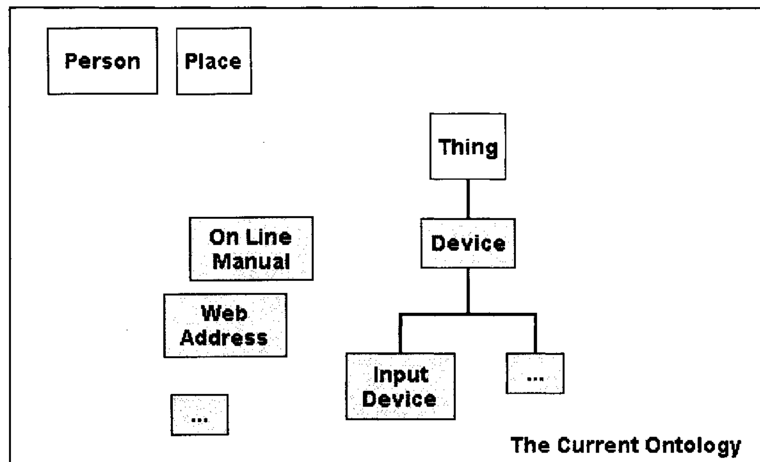


Figure 93. The system ontology "Device" ontology is added.

The next step is to add the "Library Resources" ontology to the current ontology. The Library ontology defines a hierarchy of resources (books, articles, etc.), along with other concepts such as **Author** and **ISBN** identifier. By design, the **LibraryResource** concept is intended to be a sub-concept of **Thing**. In addition, some of the concepts in the Library ontology are defined to be synonyms or related to concepts in the other ontologies, in a set of axioms explained below. As above, the Library ontology is encoded in a DAML+OIL XML file, which defines all the Classes, Properties, and relations of the ontology.

The **LibraryResource** concept is intended to be a sub-concept of **Thing**. Therefore, the composition algorithm must be given the hint to 'link ("library.daml#LibraryResource", "ppt.daml#Thing")'. In addition, some of the concepts in the Library ontology are defined to be synonyms or related to concepts in the other ontologies. These will be added as axioms in a final step.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The input to the composition is the URL of the Library ontology, the link to define the relationship between **LibraryResource** and **Thing**, and the URL of one or more Ontologies containing Axioms.

The classes from the Library ontology are added to the combined ontology. This is accomplished by reading the ontology. The DAML+OIL XML file is parsed to create an instance of *uk.ac.man.cs.img.oil.data.Ontology*. Then the input ontology is merged into the current ontology using the *Ontology.include()* method.

The intended relationship between **LibraryResource** and **Thing** is implemented as a link. The link is added to the Ontology, to define **LibraryResource** as a sub-class of **Thing**. Also, an axiom is added to assert that **LibraryResource** is disjoint from **Device**. Figure 94 shows the composed ontology at this stage.

The third step is to define trans-ontology relations, i.e., conceptual links between the two ontologies. An ontology is created with "axioms" that define relations between concepts in the two ontologies. This ontology is composed into the combined ontology using the same process described above.

In this example, three relations are defined as axioms. These three axioms are encoded in one or more DAML+OIL XML files. Figure 95 gives an example of a DAML+OIL file to define the relationships defined above.

1. The Devices ontology defines a concept called **OnLineManual**, which we define to be a sub-class of a **WebPage** in the Library ontology. (lines 10-14)
2. In the Devices ontology the **On Line Manual** is defined to have a **WebAddress. WebAddress** is defined to be a synonym for the Library ontology concept, **URL.** (lines 15-19)
3. The Library ontology has a concept called **Author**. This should be a defined as a sub concept of **Person** in the PPT ontology. (lines 25-29)

This file is composed into the current ontology using the same process as above. Figure 96 shows the combined ontology with these axioms included.

If no errors occurred, the combined ontology, including the axioms, is made the current system ontology. The previous current ontology is discarded.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

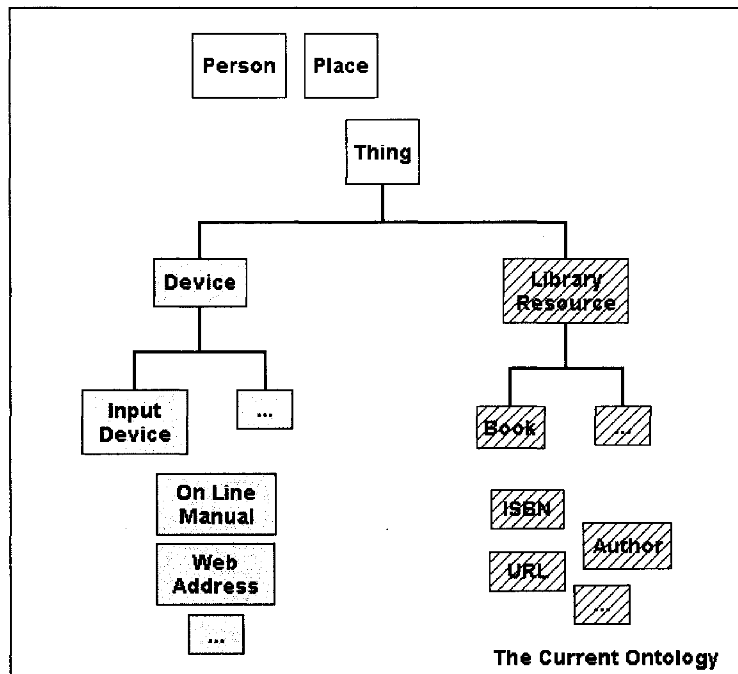


Figure 94. The "Library Resource" ontology is added to the combined ontology.

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <rdf:RDF xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
3   ...
7   xmlns:xsd="http://www.w3.org/2000/10/XMLSchema#">
8   <daml:Ontology rdf:about="">
9   </daml:Ontology>
10   <daml:Class rdf:about="http://somewhere.net/ontologies/devices.daml#OnLineManual">
11      <daml:subClassOf>
12         <daml:Class rdf:about="http://somewhere.net/ontologies/library.daml#WebPage"/>
13      </daml:subClassOf>
14   </daml:Class>
15   <daml:Class rdf:about="http://somewhere.net/ontologies/devices.daml#WebAddress">
16      <daml:sameClassAs>
17         <daml:Class rdf:about="http://somewhere.net/ontologies/library.daml#URL"/>
18      </daml:sameClassAs>
19   </daml:Class>
20   <daml:Class rdf:about="http://somewhere.net/ontologies/library.daml#URL">
21      <daml:sameClassAs>
22         <daml:Class rdf:about="http://somewhere.net/ontologies/devices.daml#WebAddress"/>
23      </daml:sameClassAs>
24   </daml:Class>
25   <daml:Class rdf:about="http://somewhere.net/ontologies/library6.daml#Author">
26       <rdfs:subClassOf>
27          <daml:Class rdf:about="http://somewhere.net/ontologies/PPT.daml#Person"/>
28       </rdfs:subClassOf>
29   </daml:Class>
30 </rdf:RDF>
```

Figure 95. DAML Axioms added to the ontology.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

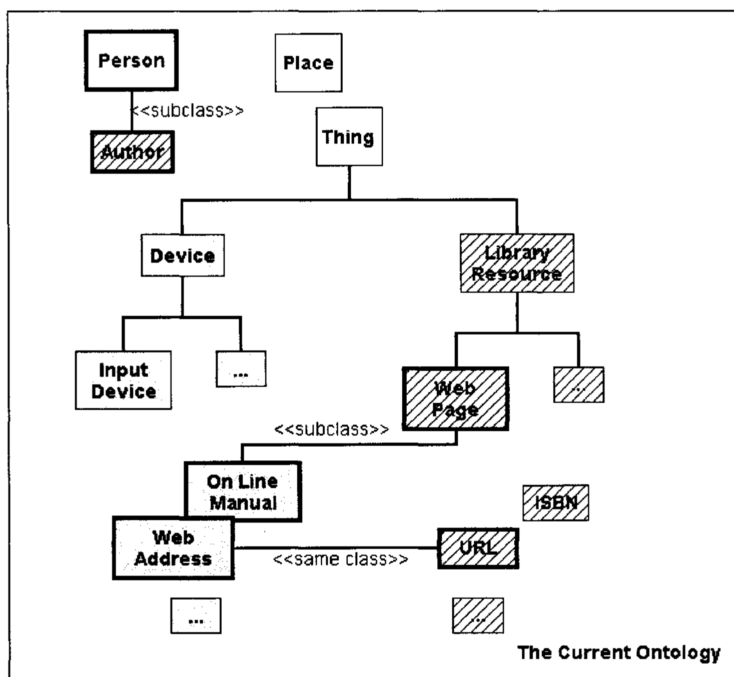


Figure 96. Additional relationships added to the combined ontology.

As this example shows, the composed ontology is not limited to be the union of the input ontologies, it contains new relationships introduced by axioms. Furthermore, the automated reasoning propagates the implications of the trans-ontology correspondences, assuring logical consistency and discovering implicit subsumption and equivalence. For example, with the combined ontology shown here, a query about "Devices#OnLineManual" will automatically discover instances of "Library#Web Page".

5. Implementation of Semantic Queries

The prototype Ontology Server implemented the semantic query architecture and algorithms described in Chapter 4.

The three stages of a Semantic Query were defined as:

1. A query to the Ontology Service to discover classes that may satisfy the query (i.e., "semantically equivalent" concepts, as defined above)
2. A query to one or more services (e.g., the Gaia Space Repository) to discover instances of these classes in the current environment
3. From the candidates, the caller or an agent selects the best choice(s)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

This section presents a prototype implementation of the first phase, a semantic match based on the current ontology.

Section 5.1 presents the query methods of the **Ontology Server**. Section 5.2 describes the two parts of the input query. Section 5.3 describes the implementation of the semantic match define in Chapter 3. The query is executed using the data structures and Knowledge Base described in section 4.2 and 4.3 of this Chapter. The results of the query are returned in several forms, which are described in Section 5.4.

5.1. The Query Interface

The Ontology Service interface was described in Section 2 (See Figure 74 and Figure 75, above). Table 31 gives details of the query methods and their parameters.

The *matchConcept* method tests if two concepts (DAML+OIL classes) match. The four *matchSet??* methods return the set of classes that match the input class *Cname*. The optional argument *Qdesc* defines one or both concepts along with other constraints on the query. Each method has alternative formats for inputs and outputs; the *options* input parameter selects between alternatives. The four methods that return sets are very similar, differing mainly in the input or output formats. This section will focus on the *matchSetXMLRanked* method, other methods will be mentioned only where they different significantly.

5.2. Description of the Query (Input)

A query has two parts,

- *Cname* – the name of a DAML+OIL class or instance to be matched. The names must conform to the DAML+OIL standard, which uses the Resource Description Framework (RDF) XML [243] standard for naming classes.
- *Qdesc* – (optional) a DAML+OIL XML file describing one or more classes, relations, and properties.

The goal of the query is to find the set of classes that match *Cname*, given the facts in the KB plus the facts specified in the *Qdesc* (if any).

Essentially, the query description (*Qdesc*) defines hypothetical facts surrounding the query, which can be written in the same language as the ontology. The query description is a DAML+OIL XML document that may contain any legal DAML+OIL statements. Then the query description is composed into the current ontology using the method described in section 3,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

to create a temporary ontology. The query is executed using the temporary ontology, as explained in section 5.3 below..

The query description has several important uses. First, it is a mechanism for defining the exact properties of the class to be discovered. The query can be constructed by creating a DAML+OIL description of a dummy class with the desired properties (*subClassOf*, *hasClass*, etc.), and then querying for the name of the dummy class. The description defines not only the name of the class, but its parent class(es), and properties. In this way, a complex query can be constructed by example.

Table 31. Summary of the Query Methods (From the CORBA interface)

Method	Parameters
boolean matchConcept(in string C1, in string C2);	Input: *C1*, *C2*: DAML+OIL class names Returns: *true* if *C1* matches *C2*
long matchSet(in string Cname, out StringSeq hits, in long options);	Input: *Cname*: the concept to match, a DAML+OIL class name *options*: 0 = match all super classes 1 = match only direct superclasses 2 = return list sorted Output: *hits*: List of concepts (DAML+OIL class names) Returns: Number of matches
long matchSetFromXML(in string Cname, in string Cdesc, out StringSeq hits, in long options);	Input: *Cname*, *options*: same as above *Cdesc*: DAML+OIL XML to describe the concepts of the query. Output: same as above Returns: Number of matches
long matchSetXML(in string Cname, in string Cdesc, out string hits, in long options);	Input: same as above Output: *hits*: A list of the concepts that match, formatted as a DAML+OIL XML document. Returns: Number of matches
long matchSetXMLRanked(in string Cname, in string Cdesc, out ResultSet hits);	Input: same as above Output: *hits*: A list of records, each record is a concept that matches, with the "degree of match". Returns: Number of matches.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Alternatively, the DAML+OIL can be viewed as a definition of the terminology used in the query. The DAML+OIL in the *Qdesc* may contain any information that is relevant to the query and may refer to any DAML+OIL ontology. This may be used, for instance, to define the query using the terminology of the caller, which ideally will be mapped automatically to the terminology of the system ontology. For example, the caller may use the term "Slide Show" (with appropriate definitions), which the Ontology Service might be able to deduce is equivalent to "Power Point Presentation" in the composed ontology.

Third, the DAML+OIL description can be used to define hypothetical conditions. For example, the DAML+OIL can define a combination of classes with certain properties. The query checks that this state is possible (logically consistent with the current KB), and then answers the query. In this case, the *Qdesc* describes a set of hypothetical facts, *H*, and the query is something like, "*Suppose that H is true, what classes match C?*"

For example, to discover sources of water in the current building, the *Qdesc* might be a DAML+OIL file that defines a specific kind of WaterFountain, e.g., wheelchair accessible. Then the query would query this concept to discover all classes that match, including wheelchair accessible faucets, and so on.

A simple query to find classes that match 'water fountain' could be:

Cname = "http://www.onto.net/things.daml#WaterFountain"

Qdesc = null

This query should return all the classes similar to **WaterFountain**, which might include sinks, vending machines, and so on.

To frame a more sophisticated query, a Qdesc would be created to describe the characteristics of the desired target, e.g., a source of water that is wheel chair accessible. Figure 97 shows an example of what such a *Qdesc* might look like.

The query would be:

Cname = "http://www.onto.net/things.daml#AccessibleWaterFountain"

Qdesc = <DAML+OIL XML from Figure 97>

This example illustrates that this is a rudimentary form of *query by example.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <rdf:RDF xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
3   xmlns:dc="http://purl.org/dc/elements/1.1/"
4   xmlns:OilEd="http://img.cs.man.ac.uk/oil/OilEd#"
5   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
6   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
7   xmlns:xsd="http://www.w3.org/2000/10/XMLSchema#">
8   <daml:Ontology rdf:about="">
9   </daml:Ontology>
10  <daml:Class rdf:about="http://somewhere.net/things.daml#BuildingServices">
11  </daml:Class>
12  <daml:Class rdf:about="http://somewhere.net/things.daml#SourceOfWater">
13    <rdfs:subClassOf>
14      <daml:Class rdf:about="http://somewhere.net/things.daml#BuildingServices"/>
15    </rdfs:subClassOf>
16  </daml:Class>
17  <daml:Class rdf:about="http://somewhere.net/things.daml#WaterFountain">
18    <rdfs:subClassOf>
19      <daml:Class rdf:about="http://somewhere.net/things.daml#SourceOfWater"/>
20    </rdfs:subClassOf>
21  </daml:Class>
22  <daml:Class rdf:about="http://somewhere.net/things.daml#WheelChairAccessible">
23    <rdfs:subClassOf>
24      <daml:Class rdf:about="http://somewhere.net/things.daml#Accessibility"/>
25    </rdfs:subClassOf>
26  </daml:Class>
27  <daml:Class rdf:about="http://somewhere.net/things.daml#Accessibility">
28  </daml:Class>
29 </rdf:RDF>
```

Figure 97. Sample DAML+OIL XML to define a specific kind of water fountain.

5.3. Query Execution

A query *matchSetXMLRanked(Cname, Cdesc, hits*) is implemented in the following steps:

1. If *Qdesc* is defined, the ontology in *Qdesc* is composed into the current system ontology to create a temporary ontology. This is done by the same process as described in section 3, except the combined ontology does not permanently replace the current system ontology.
2. If the temporary ontology (the current ontology with the additional facts in *Qdesc*) is valid, then the hypothesis is logically possible, the proposed query may be executed.
3. A query for the specified class is executed, implementing the semantic match defined in Chapter 3.
4. After the query is completed, the hypothesis is withdrawn, i.e., the KB is restored to its original state.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

These steps are described in the following sections.

5.3.1. Load and Validate the Query Description

If a query description is defined, a temporary ontology is created to define the conditions of the query.

First, the current ontology is saved to a file by calling *saveOntology*. This ontology will be restored after the query.

Second, the query description is added to the current ontology, using the composition algorithm described in section 3. This step uses exactly the same mechanism used to compose new ontologies or axioms into the current ontology.

If the temporary ontology is not valid, then the query conditions create a contradiction. In this case, the query the match set is empty, and the query returns "false". If the composed ontology is valid, the query is executed against the temporary ontology.

If no query description was given, this step is skipped and the query is executed against the current ontology and the KB.

5.3.2. Implementation of the Semantic Match Algorithm

The goal of the query is to discover the set of concepts (DAML+OIL classes) that match the input concept, *Cname*, according the definition in Chapter 3 above.

First, *Cname* is looked up in the ontology. If the class is not found (i.e., there is no such class), the result is the empty set. Note that it is possible to query for a concept not represented by a class in the ontology by defining the concept as a class in the *Qdesc*, and querying the synthesized class (i.e., *Cname* can be some class defined in the *Qdesc*).

Given that *Cname* exists, the result set is the union of the five sets, corresponding to the degrees of match, one of *{ NO, SUBSTITUTE, COPARENT, SUBSUME, PLUGIN, EXACT }*.

As discussed in section 2, the current ontology is represented by an instance of the Java class *uk.ac.man.cs.img.oil.data.Ontology.* Assuming that the ontology has been validated, as described in section 4.2, the data structure represents a "cache" of the subsumption relationships implied in the Knowledge Base. Therefore, the *uk.ac.man.cs.img.oil.data.Ontology* data structure can be use to discover the concepts that meet the criteria of the match, simply by traversing the graph to discover the ancestors, children, and siblings of a class. There is no need to query the KB, except to test the consistency of two classes.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

First, the EXACT matches are discovered. For a give class, *Cname*, the set of equivalent classes may be cached in the Ontology data structure; if not, it is computed by iterating through all the classes, and testing subsumption. For two classes *C* and *D*, *D* is an exact match if and only if *(KB.subsumes(C, D) AND KB.subsumes(D, C))*. Trivially, the *Cname* is an EXACT match for itself, and may be omitted.

Second, the PLUGIN matches are determined. For a class *Cname*, the set of PLUGIN matches is defined to be all sub-classes of *Cname*. The sub-classes of *C* are obtained from the class graph of the Ontology data structure. Third, the SUSUME matches are discovered. For a class *Cname*, the set of SUBSUME matches is defined to be the ancestors of *Cname* in the class graph. The super classes are obtained by traversing the class graph of the Ontology. By convention, the class *Cname* is omitted from these sets.

Fourth, the COPARENT matches are discovered. For a class *C*, the coparents can be determined by navigating the class graph:

1. Find all the parents of *C*.
2. For each $d \in$ *parents(C)*, find all the sub classes of *d*.

Finally, the SUBSTITUTE matches are found. This is the set defined in Chapter 3, Section 7. First, the set of candidate classes are discovered. For a class *C*, this set is:

1. Find all the ancestors of *C*. This set may be limited by a pruning rule, e.g., to include only parents or grandparents of *C*.
2. For each $d \in$ *ancestors(C)*, find all the sub classes of *d*.

A give class may appear in this set several times, but should be tested only once.

For each candidate, *D*, the consistency test is performed by the *OntoKB.compatible()* method, as describe in Chapter 3.

The result set is the union of these five sets. A class may appear in more than one set, if it meets more than one of the criteria. The result set must be processed to present the answer to the caller. Section 5.4 discusses several alternative formats for the output.

5.3.4. Clean up and Withdraw the Query Description

After the query is completed, the ontology must be restored to its original state.

If the *Qdesc* was defined, then the assertions of the query description must be withdrawn from the Knowledge Base. The Ontology Server implements this by discarding the temporary ontology, and reloading the previous ontology from the snapshot.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.4. Result Sets and Information Returned to the Caller

There is no single correct way to return the results of the query to the calling program. The results must be filtered depending on the requirements of the caller. It is difficult to define a single general run, because different uses require different information from the result.

In some cases, duplicates should be retained, in other cases they should be discarded. Sometimes "trivial" results (e.g., the input class itself) should be discarded, sometimes they should be retained. The results might be sorted alphabetically or in order of the degree of match, or according to some task-specific criteria.

The prototype implementation provides a few simple alternatives, with the assumption that the caller must process the result according to its own needs.

The *matchSetXMLRanked* method returns a *ResultSet* object. The *ResultSet* is a list of zero or more *ResultRec* objects, each of which is the name of a DAML+OIL class and a degree of match, one of *{ NO, SUBSTITUTE,COPARENT, SUBSUME, PLUGIN, EXACT }*. Several alternative queries return the result set in different formats: as a list of class names, as string, or as XML. The set of matches is the same in each case, the difference is the format of the results.

The result set may contain quite a few "trivial" matches, e.g., for class *C*, the result includes *C* and may include ancestors of *C* all the way to the root of the graph. While these classes are technically matches (e.g., "**Thing**" may match nearly every concept), they are usually not useful answers. On the other hand, these matches may provide evidence for why certain other items were matched (e.g., because of a common ancestor) which may help some applications to interpret the result.

5.5. Discussion

The **Ontology Server** cannot give perfect answers, it can only answer based on the information in the ontologies and the Knowledge Base. The information is often incomplete, which may cause the result may contain false hits and/or misses. When two concepts are given simple definitions with only a few attributes, they might be indistinguishable except by their names. In this case, based on the limited information in the KB they may be considered equivalent by the query algorithm. This may or may not be a valid result in the real world.

In addition to the set of classes that match the query, the caller may require additional information, especially why the class matched (e.g., the reasoning that lead to its inclusion) and who says so (e.g., what ontologies were used). This amounts to providing the caller with an

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

explanation of the reasoning that resulted in the answer. This is an extremely difficult problem for any automated reasoning system, and this study did not address this challenge.

6. Summary

This Chapter presented a prototype Ontology Server which implemented the algorithms defined in Chapter 4. The Ontology Server is a CORBA service, which manages ontologies written in DAML+OIL XML. The interfaces, design, and main algorithms were presented in detail.

The Ontology Server reads and parses DAML+OIL XML files to create and update a system ontology. The Ontology Service used the CORBA FaCT Server as the KB, and uses the Java OIL package to parse DAML+OIL and translate into Description Logic. The KB is used to prove facts about DAML+OIL XML files, including logical consistency and subsumption.

The Ontology Server implements the composition algorithm defined in Chapter 4. The Ontology Server represents the current ontology as an instance of the Java class *uk.ac.man.cs.img.oil.data.Ontology*. Additional concepts are input as a DAML+OIL XML file, which is added to the system ontology and proved logically consistent (valid).

The Ontology Server implements the semantic match defined in Chapter 3. The Ontology Server defines an interface for queries using DAML+OIL XML. The queries are resolved using the semantic match algorithm defined in Chapter 4. The results are returned in several formats.

The semantic match answers queries based on the current system ontology, i.e., the result of the cumulative effect of the composition algorithm. The query is implemented by a combination of operations on data structures and queries to the Knowledge Base. The data structures are essentially a cache of the state of the Knowledge Base.

The prototype was evaluated in several experiments, reported in Chapter 7.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 7. Evaluation of the Prototype

1. Introduction

The preceding chapters presented example ontologies and a prototype implementation of an Ontology Service. This chapter presents an evaluation of the correctness and performance of the prototype.

The prototype was implemented as a stand alone test system. Subsequently, it was ported to the Gaia Pervasive Computing Environment. Section 2 discusses this pragmatic demonstration of the usability of the prototype software.

The test system was evaluated in depth. Section 3 describes the methods used. Section 4 discusses the correctness of the implementation. Section 5 and 6 present performance measurements.

Section 7 summarizes the findings.

2. General Design and Usability: Integration into Gaia

The prototype **Ontology Server** was implemented as a CORBA service written in Java, as described in Chapter 6. The initial prototype was a standalone service with custom clients.

In collaborative work, the **Ontology Server** was ported to the Gaia Pervasive Computing Environment [151, 201-203]. The port was not difficult, requiring a small amount of code to integrate into the Gaia system. This experience confirmed the basic design and showed that the prototype could be used in a Ubiquitous Computing Environment.

In the Gaia environment, an instance of the Ontology Server maintains a single, cumulative "current ontology" for a Gaia Active Space. Each Active Space has one Ontology Server running in it. Other entities in Gaia contact the Ontology Server to get descriptions of entities in the environment, meta-information about context or definitions of various terms used in Gaia.

In the Gaia implementation, the unified ontology maintained by the Ontology Server serves as a logical schema used by all the different services. Other services provide information about the specific entities that are available. The Gaia Space Repository maintains information

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

about the entities in the space at any time [205]. Instances of context information are distributed among different sensors and other entities that use context [199].

Services, applications, and entities in the Gaia system contact the Ontology Server to discover the classes and relations of entities that may be found in the Active Space. To illustrate this use, an Ontology Explorer was implemented which allows users to browse and search the ontologies in the space [151]. The Ontology Explorer allows users to interact with other entities in the space through it. The interaction with other entities is governed by their properties as defined in the ontology. This Ontology Explorer is similar to a class browser, except it has information about all the entities of the system, not just the software classes.

3. Evaluation Criteria: Correctness and Performance

Three aspects of the prototype implementation were evaluated:

1. Example ontologies
2. The composition algorithm
3. The query algorithm

This section summarizes the analysis of correctness and performance for these features of the implementation

3.1. Evaluating Ontologies

The *correctness* of an ontology can be assessed in several ways.

An ontology is correct only if it accurately reflects the concepts of the domain. This evaluation must consider whether the concepts of the domain model(s) are correctly and completely represented. Ultimately, this can only be a subjective judgment by the domain experts. The might be done through independent review by multiple judges. In the case of a scientific or technical domain, a peer review process would be applied. In other cases, a standards body, such as the IEEE, would validate the ontology. These types of validation were not attempted for the example ontologies used in this project.

When the ontology is based on a formal model, it is important to show that all the facts deduced from the ontology are true in the model. Details of such a proof depends on the model. The example models used in this study had only rudimentary formal constraints, which were evaluated by the test queries discussed in this chapter.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Assuming the ontology adequately represents the domain knowledge, the XML encoding of the ontology can be shown to be syntactically correct and logically valid. Tool such as OilEd [8, 178], Protégé [170, 171], or Ontoedit [221] automatically create syntactically correct XML. These tools can also validate an ontology. For example, the OilEd tool can validate ontologies with the CORBA FaCT Server [8], the Racer server [94], or other Knowledge Base.

The *performance* of ontologies is measured by evaluation of performance of algorithms for different ontologies and queries, as discussed below.

3.2. Evaluating Implementation of the Algorithms

The key algorithms include the validation and composition algorithms presented in Chapter 3. The *correctness* of the implementation can be evaluated by performing experiments and inspecting the results to confirm that the answers are correct and the Knowledge Base has the correct contents.

The *performance* of the implementation can be evaluated by incrementing the Ontology Service, and measuring the time to validate and compose sample ontologies.

The Ontology Service is a client-server architecture. The client initiates the operation with a CORBA method invocation to invoke one of the operations described in Chapter 6. The client's message contains the parameters of the method, possibly including URLs of one or more ontologies (XML files).

Each operation has several steps, including:

- Client processing
- CORBA messaging to server
- Network latency
- Server processing

The Ontology Service processes the request as described in Chapter 5. The implementation of server processing has several major components that contribute to the overall performance:

- Processing the input arguments
- Calling the Knowledge Base (via a local socket or CORBA connection)
- Miscellaneous memory operations
- Writing snapshots to disk and other overhead
- Returning results

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

This chapter presents measurements to estimate the run time for these computational steps.

3.3. The Test Environment and Methodology

The prototype Ontology Service was evaluated in a series of experiments using the FaCT server. The prototype was instrumented to collect elapsed time for several major operations. The instrumented server was tested with a series of example ontologies and queries.

There were two independent variables:

1. input ontologies
2. test queries

The dependent variables were measurements of elapsed time, and recall/precision for queries.

The Ontology Service was instrumented to measure the elapsed time for sections of the code. The measurements were collected on a small test system:

- Pentium 3 with 256MB memory
- All CORBA connections are through local sockets, no network link was used.
- XML files were read from local disk, not from network URLs
- The test client, Ontology Server, and CORBA FaCT server all run on same system.

The time was measured by calling the standard Java method, *System.currentTimeMillis()*. The times reported are the average of five runs.

Clearly, the absolute performance of the Ontology Service was limited by the test platform. Using a faster system with more memory would speed up all the operations of the server. Distributing the client and servers to dedicated nodes would decrease the processing time, but would increase the latency of the CORBA calls.

For consistency, all the measurements used the FaCT server as the Knowledge Base. As discussed in Chapter 6, the prototype can be configured to use alternative Knowledge Bases. In future work, it will be possible to compare the performance of different KBs.

3.4. Evaluating Queries and Query Answering

A query algorithm can be evaluated on several criteria:

1. correctness (i.e., correctly implements the algorithm, the answers are correct)
2. subjective value (the answers are non-trivial)
3. efficiency (run time and memory)

For any query-response protocol, the *correctness* of a semantic query is at least somewhat subjective: the correctness of the result partly depends on the intention of the questioner and the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

interpretation of the result. The results of the queries can be examined to (informally) determine that they make sense, i.e., the answer is reasonable. In addition, the answers can be reviewed to determine if they seem non-trivial, e.g., something that would not be returned by just matching *strings*.

The effectiveness of a query can be measured with a *precision/recall* metric used in information retrieval. The *performance* of the query algorithm was assessed by measuring the run time of test queries.

The Ontology Service was tested with several test ontologies. In each case, one or more ontologies were loaded into the Ontology Service, and several queries were executed. The results of a query were a set of classes, optionally ranked according to "degree of match".

The input to the experiments is a sequence of ontologies, composed to create a current system ontology as described earlier in this chapter. The individual ontologies vary from zero to 24 concepts. As more ontologies are composed, the current system ontology grows.

The first case is an example from Gonzalez-Castillo et al. ([80], page 8). The second example used an ontology developed for the Gaia Pervasive Computing Environment, as reported in earlier work [150, 201, 203]. For a third case, the ontologies described in Chapter 5 and 6 were composed.

The final experiment built a moderate size ontology (136 concepts) through a sequence of compositions, composing in 8 ontologies, plus linking axioms, for a total of 10 steps, excluding the initialization.

In the case of the composition algorithm, the ontology is input as the URL of one or more ontologies (DAML+OIL XML files) which are to be loaded by the server. In the case of a query, the input is the target of the query (the name of a concept) and optional *Qdesc* (DAML+OIL XML, as described in Chapter 6). The result is a list of matches.

In each case, the time to load, verify, and compose ontologies was measured. Test queries were executed to the Ontology Service. The results were evaluated, and the time to answer the queries was measured. Table 32 lists the test input.

Chapter 3 gave the definition of "semantic match", along with intuitive justification and interpretation. Chapter 5 presented a prototype implementation.

The effectiveness of a query can be measured with a precision/recall metric used in information retrieval [134, 262]. These statistics indicate the proportion of the items in the result

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

set that correctly answer the question (precision), and the proportion of correct answers that are included in the result set (recall). Figure 98 gives the definition of these statistics. These measures are based on subjective judgment of what the correct results should be for a give query and Knowledge Base.

Table 32. Summary of Input Ontologies.

Ontology	Source	Queries
"Device Ontology"	Adapted from Gonzalez-Castillo et al. [80]	Example from the paper, Query each concept
Gaia Applications	[150, 201, 203]	Query each concept.
Ontologies, to be composed 1. People, Places, and Things 2. Devices 3. Library	See Chapter 5	Query each concept.
Larger ontology (up to 136 concepts): 1. People, Places, and Things 2. Devices 3. Library 4. Duplicates of the Library ontology (to increase size)	See Section 5.	Query each concept.

For a set of results, three numbers are defined.
#results = the number of items (class names) in the result set.
#correct = the number of items (class names) in the ontology that correctly match the query. I.e., the number of elements in the known correct answer.
#hits = the number of correct items in the result set.
Two statistics are defined:
Precision = *#hits / #results*
Recall = *#hits / #correct*
E.g., see Korfage [134].

Figure 98. Definition of Precision and Recall used in this section.

In this study, these measures are computed by examining each query, to determine the (subjectively) correct answer: i.e., the set of classes that should match the given query. The query result is compared to the subjectively correct answer to determine the hits (classes correctly matched) and false positives (classes incorrectly matched). The two statistics are ratios, defined

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

in Figure 98. It is important to emphasize that these are subjective measures: the correctness of an answer is a judgment.

4. Results 1: Correctness

This section summarizes the investigation of the correctness of the implementation of the composition and query algorithms.

4.1. Correctness of the Composition Algorithm

The composition algorithm is implemented as a series of operations that depend on the logical mapping from DAML+OIL to Description Logic (e.g., see [117]). The Ontology Server depends on the implementation Java OIL package and CORBA FaCT server to be correct [8, 178].

The input cases were used to hand check the behavior of this software. The ontologies presented in Chapter 5 were loaded and verified into the Ontology Server. The checkpoint facility was used to save snapshots of the state of the Knowledge Base (as DAML+OIL XML files) at each step of the algorithms. Visual inspection and browsing with the OilEd tool confirmed that the load and composition operations created the correct Knowledge Base [8].

As an additional cross-check, the input ontologies and snapshots were validated using the Racer Knowledge Base [94]. The results were identical with those of the Ontology Server and the OilEd tool: no errors were detected with the second Knowledge Base.

4.2. Correctness of Queries

This section presents the results of the analysis of the queries for size cases. Each case consists of one or more ontologies that are composed to create the system ontology, followed by a series of queries. Precision and recall statistics were computed for the queries. Table 33 lists the test cases used in the experiments.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 33. Test cases used to evaluate the query algorithm.

Ontology	Queries	Notes
Case 1: "Device Ontology" from Gonzalez-Castillo et al. [80]	Example from the paper.	Demonstrates the correctness against a published definition.
Case 2: "Device Ontology" from Gonzalez-Castillo et al. [80]	All concepts in the ontology.	Check the ontology against the published paper.
Case 3: Gaia Applications [150, 201, 203]	Example from the papers.	Demonstrate that the implementation provides a non-trivial service.
Case 4: People, Places, and Things, Devices, and Library, as described in Chapter 5.	All concepts in the ontology.	Demonstrate the correctness and effect of the composition algorithm.
Case 5: The same Ontology as Case 5, with specific concepts are composed into the ontology.	Printer and Fax, as discussed in Chapter 5.	Demonstrate details of query algorithm and composition algorithm.

4.2.1. Case 1: Example from Gonzalez-Castillo et al.

The first example considered is the example from Gonzalez-Castillo et al. [80]. Figure 99 shows a sketch of the class hierarchy they defined, with some details omitted to save space. To test the implementation, this example was encoded in DAML+OIL and loaded into the Ontology Service. The DAML+OIL XML was shown in Chapter 5.

Gonzalez-Castillo et al. present a query to "find all services that match 'SERV5'" ([80], page 8). Several variations of this query were created, and the results examined. Then each class in the ontology was used as a query, and the results rated for precision and recall.

The first query is a class called **SERV5a**, which is an exact duplicate of **SERV5**, but without its "children", i.e., **SERV5a** has the same definition as **SERV5**, but is not a super class of **SERV4**. Figure 100 shows the DAML+OIL for this query description for this class. The query and result set is illustrated in Figure 101. In the figure, the set of classes that match the query are shaded. The set of matches is precisely the set specified in [80], except it includes some trivial cases, **Service-Description**, and **computer** (trivial answers are hatched).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

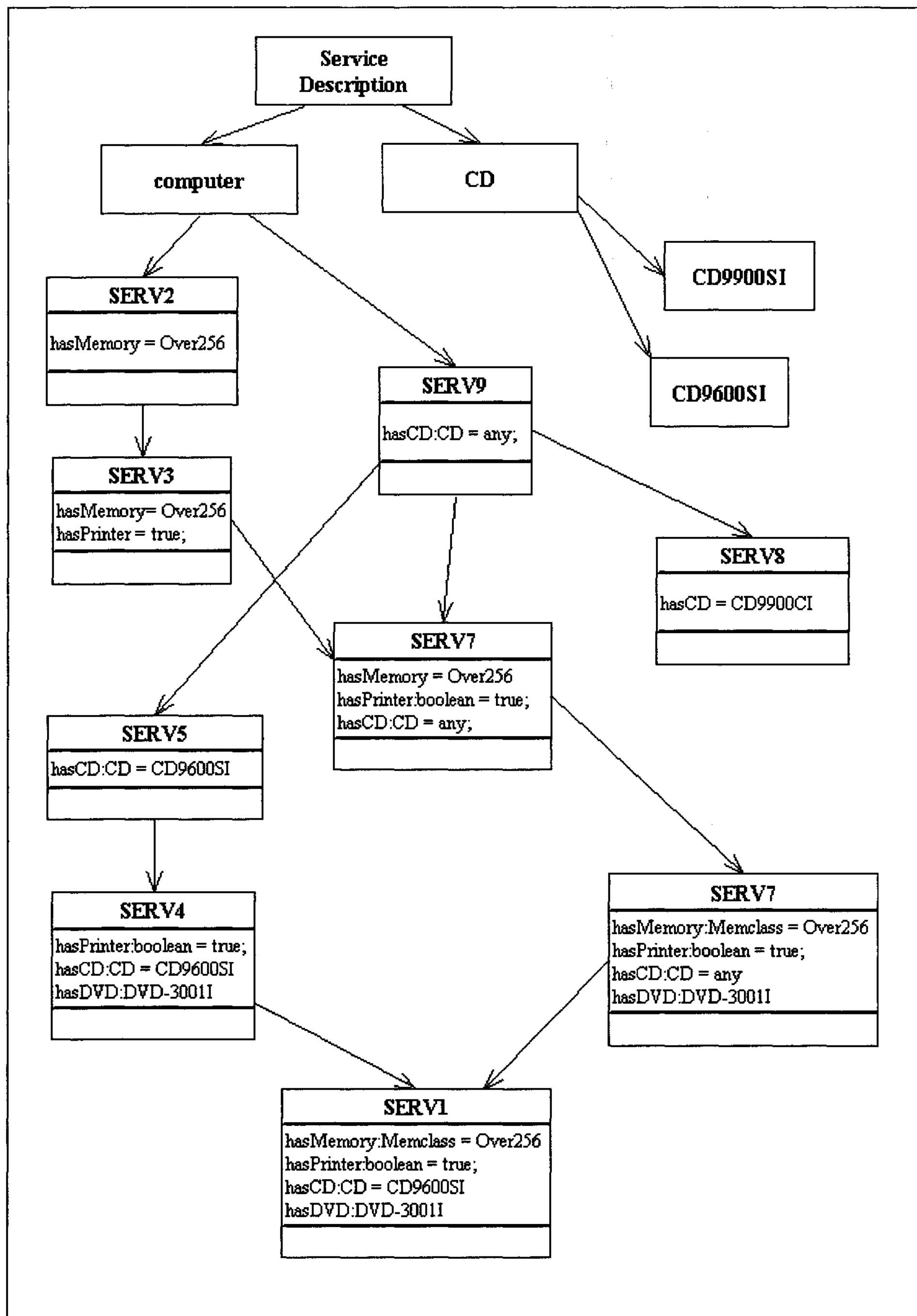


Figure 99. Hierarchy of classes, as in [80].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
 1 <?xml version="1.0" encoding="ISO-8859-1"?>
 2 <rdf:RDF xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
 3   xmlns:dc="http://purl.org/dc/elements/1.1/"
 4   xmlns:OilEd="http://img.cs.man.ac.uk/oil/OilEd#"
 5   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 6   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
 7  xmlns:xsd="http://www.w3.org/2000/10/XMLSchema#">
 8   <daml:Ontology rdf:about="">
 9   </daml:Ontology>
10   <daml:Class rdf:about="http://somewhere.net/sd.daml#SERV5a">
11     <rdfs:subClassOf>
12         <daml:Class rdf:about="http://somewhere.net/sd.daml#SERV9"/>
13     </rdfs:subClassOf>
14     <rdfs:subClassOf>
15       <daml:Restriction>
16       <daml:onProperty rdf:resource="http://somewhere.net/sd.daml#hasCD"/>
17           <daml:hasClass>
18             <daml:Class rdf:about="http://somewhere.net/sd.daml#CD9600SI"/>
19           </daml:hasClass>
20       </daml:Restriction>
21     </rdfs:subClassOf>
22   </daml:Class>
23 </rdf:RDF>
```

Figure 100. The DAML 'query' for 'SERV5a', identical to 'SERV5'.

Table 34. Matches for SERV5a

Degree of Math	Class
EXACT	<empty> (SERV5a omitted)
PLUGIN	<empty>
SUBSUME	SERV9 (computer, Service Description, omitted)
INTERSECT	SERV1, SERV4, SERV5, SERV6, SERV7 (SERV5a omitted)

Table 34 shows which criteria were matched (i.e., the *degree of match*) for each of the classes in the result. No classes are EXACT matches to **SERV5a**. (**SERV5a** was excluded because it is trivial.) **SERV5** is not an EXACT match for **SERV5a**, because the subclasses are not the same. There are no PLUGIN matches because **SERV5a** has no subclasses.

There is one SUBSUME match, **SERV9** (ignoring trivial results). This result is correct, because **SERV9** is a super class of **SERV5a**. There were six INTERSECT matches: **SERV4**, **SERV5**, **SERV1**, **SERV6**, **SERV7**. These classes are subclasses of superclasses of **SERV5a**, and their intersections are satisfiable. The intersection means that there is no contradiction between '**SERV5** and the class.

While **SERV8** is also a subclass of the parent of **SERV5a**, it does not match because the intersection of **SERV5a** and **SERV8** is not satisfiable. This is correct because the *hasCD*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

property is incompatible. **SERV8** *hasCD* **CD9900CI**, while **SERV5a** *hasCD* **CD9600SI**. The query correctly excluded **SERV8**.

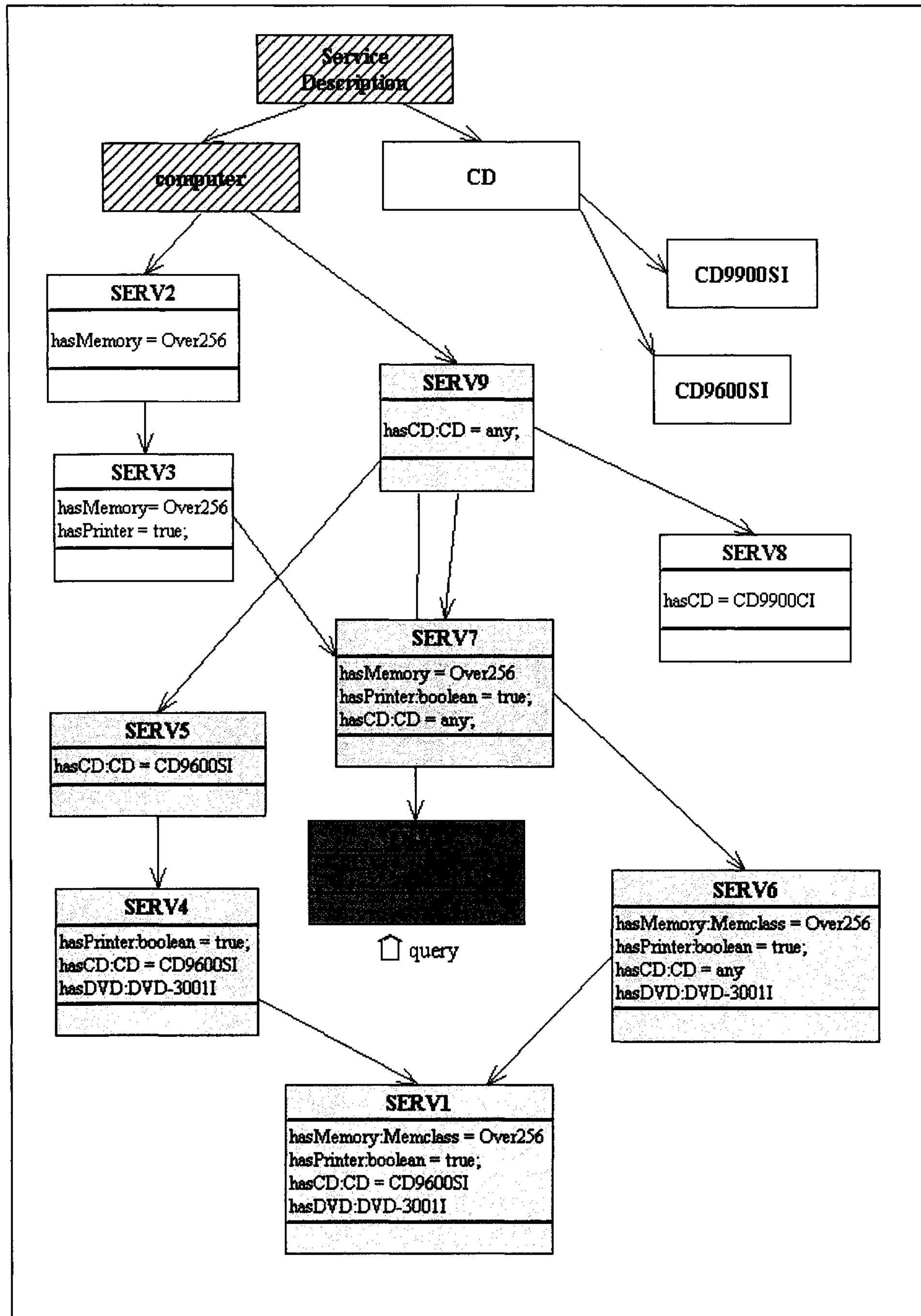


Figure 101. Illustration of the class hierarchy, with the query entered as class SERV5a. The shaded classes match the query.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

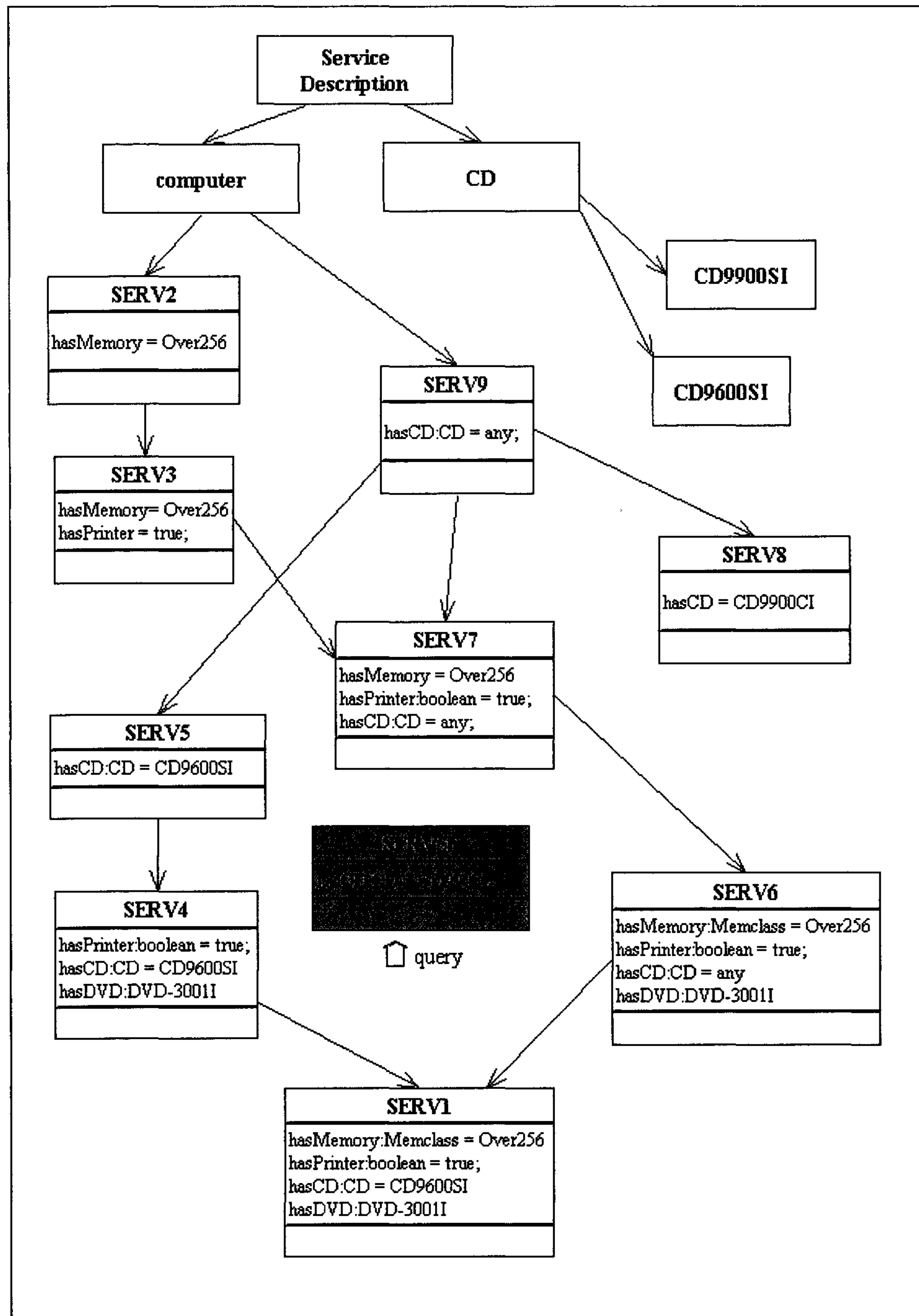


Figure 102. Illustration of the query 'SERV5b'. No classes match.

All other classes are correctly excluded from the result set.

The second test query, **SERV5b** is identical to **SERV5a**, but it is not related to any class in the ontology. This query has the same attributes as above, but the target is not specified to be a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

sub-class of **SERV9** or any other class. Figure 103 shows the DAML+OIL for this query. As expected, the query returns no matches. Figure 102 shows an illustration of this result.

```
 1 <?xml version="1.0" encoding="ISO-8859-1"?>
 2 <rdf:RDF xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
 3   xmlns:dc="http://purl.org/dc/elements/1.1/"
 4   xmlns:OilEd="http://img.cs.man.ac.uk/oil/OilEd#"
 5   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 6   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
 7   xmlns:xsd="http://www.w3.org/2000/10/XMLSchema#">
 8   <daml:Ontology rdf:about="">
 9   </daml:Ontology>
10   <daml:Class rdf:about="http://somewhere.net/sd.daml#SERV5b">
11       <rdfs:subClassOf>
12         <daml:Restriction>
13        <daml:onProperty rdf:resource="http://somewhere.net/sd.daml#hasCD"/>
14           <daml:hasClass>
15             <daml:Class rdf:about="http://somewhere.net/sd.daml#CD9600SI"/>
16           </daml:hasClass>
17         </daml:Restriction>
18       </rdfs:subClassOf>
19   </daml:Class>
20 </rdf:RDF>
```

Figure 103. DAML query for 'SERV5b'. Same properties as 'SERV5', but not related.

The third example shows how the properties are matched. The query, **SERV5c,** is similar to **SERV5**, except it is specified to have a different restriction on the *hasCD* property: the CD must be class **CD9900CI** instead of **CD9600SI**. Figure 104 shows the DAML for this query.

Figure 105 shows the result of the query. The result set includes some but not all of the classes as the results of the **SERV5a** query above. First, **SERV5** and its subclasses do not match **SERV5c**. This is correct because the property of *hasCD* does not match. Similarly,

SERV8 *does* match **SERV5c**, because the *hasCD* property is consistent.

Table 35 shows the criteria by which the classes matched. There are no EXACT or PLUGIN matches for **SERV5c**. (The class itself is omitted.) There is one SUBSUMES match, **SERV9** (excluding the trivial cases).

There are three SUBSTITUABLE matches: **SERV6**, **SERV7**, and **SERV8**. These classes do not conflict with the *hasCD* restriction to **CD9900CI**: **SERV8** is restricted to **CD9900CI**, and the others are unrestricted.

Classes **SERV5**, **SERV4**, and **SERV1** are correctly rejected. These classes have restrictions on *hasCD* that conflict with the definition of **SERV5c**. Therefore, the intersection is not satisfiable, and they are not in the result set.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
 1 <?xml version="1.0" encoding="ISO-8859-1"?>
 2 <rdf:RDF xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
 3   xmlns:dc="http://purl.org/dc/elements/1.1/"
 4   xmlns:OilEd="http://img.cs.man.ac.uk/oil/OilEd#"
 5   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 6   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
 7   xmlns:xsd="http://www.w3.org/2000/10/XMLSchema#">
 8   <daml:Ontology rdf:about="">
 9   </daml:Ontology>
10   <daml:Class rdf:about="http://somewhere.net/sd.daml#SERV5c">
11     <rdfs:subClassOf>
12       <daml:Class rdf:about="http://somewhere.net/sd.daml#SERV9"/>
13     </rdfs:subClassOf>
14     <rdfs:subClassOf>
15       <daml:Restriction>
16        <daml:onProperty rdf:resource="http://somewhere.net/sd.daml#hasCD"/>
17          <daml:hasClass>
18            <daml:Class rdf:about="http://somewhere.net/sd.daml#CD9900CI"/>
19          </daml:hasClass>
20       </daml:Restriction>
21     </rdfs:subClassOf>
22     <rdfs:subClassOf>
23      <daml:Restriction daml:cardinalityQ="1">
24 <daml:onProperty rdf:resource="http://somewhere.net/sd.daml#hasPrinter"/>
25<daml:hasClassQ rdf:resource="http://www.w3.org/2000/10/XMLSchema#boolean"/>
26       </daml:Restriction>
27     </rdfs:subClassOf>
28   </daml:Class>
29 </rdf:RDF>
```

Figure 104. DAML query for 'SERV5c', related to 'SERV5', but property conflicts.

Table 35. Matches for SERV5c

Degree of Match	Class
EXACT	<empty> (SERV5c omitted)
PLUGIN	<empty>
SUBSUME	SERV9 (computer, Service Description, omitted)
SUBSTITUTABLE	SERV6, SERV7, SERV8

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

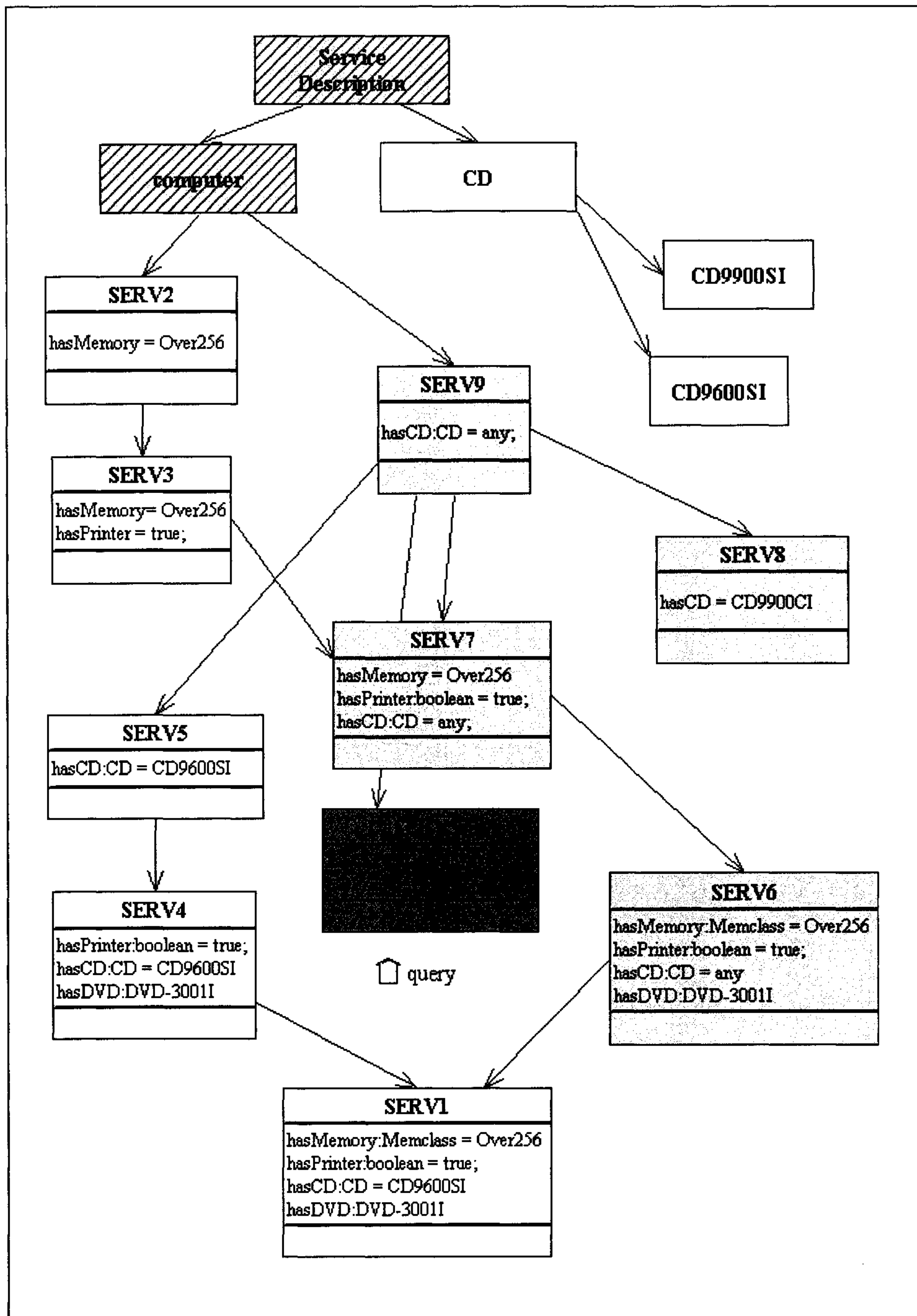


Figure 105. Illustration of the 'SERV5c'. Similar to 'SERV5a', except the 'hasCD' is set to 'CD9900CI'.

Table 36 gives the precision and recall statistics for these three cases. (The statistics are undefined when the correct result is the empty set.) The other cases have 100% precision and recall.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 36. Summary of the three examples.

Example Query	Precision (#good / #hits)	Recall (#good / #correct)
SERV5a	6/6 = 1.0	6 / 6 =1.0
SERV5b	NA	NA
SERV5c	4/4 =1.0	4/4 =1.0

4.2.2. Case 2: All Classes from the Service-Description Ontology

To fully examine this ontology, every class was used as a query. The ontology described above was loaded into the Ontology Service, then 17 queries were created, one for each class in the ontology. Table 37 gives the precision and recall for each class, including some not shown in the diagram. All the computer classes (**SERV1** through **SERV9**) give 100% precision and recall.

The query for **SERV7** illustrates an example of the *substitutable* match. Figure 106 shows the classes that match **SERV7. SERV4, SERV5**, and **SERV8** have properties that are compatible with **SERV7**. However, **SERV7** is defined as a subclass of (**SERV3** AND **SERV9**), while the other classes are subclasses of **SERV9**. Therefore, these classes do not match according to the compatibility rule.

Some of the queries have less than 100% precision. This means that the results contain *false positives*, i.e., some classes are reported to match that should not.. For example, the matches for **computer** included **CD** and its subclasses. This was judged to be an incorrect result, because **CD** is not related to **computer**. Similarly, the **CD** class matched all but one class in the ontology. These results occur because the ontology contains very little information about these classes, so they cannot be distinguished by the Ontology Service. In contrast, the **SERV** classes are carefully defined, and the queries give precise results.

Altogether, the queries give reasonable and intuitively expected results. These results suggest that the algorithm has been implemented correctly, although false positives occur when the concepts are minimally defined in the ontology.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

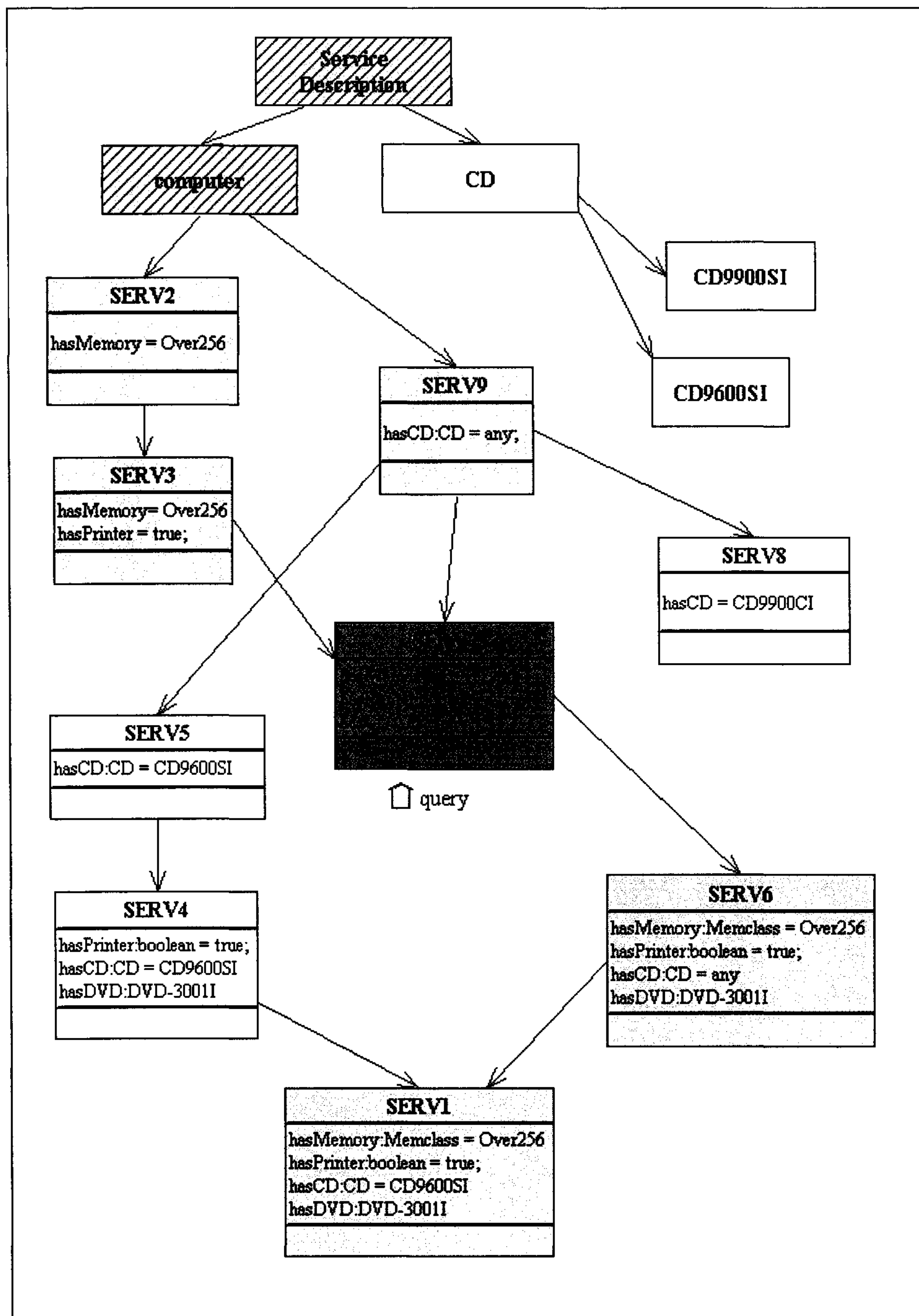


Figure 106. The matches for SERV7.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 37. Summary of the results for each query.

Query	Precision (#good / #hits)	Recall (#good / #correct)
SERV1	10/10	10/10
SERV2	11/11	11/11
SERV3	7/7	7/7
SERV4	6/6	6/6
SERV5	8/8	8/8
SERV6	8/8	8/8
SERV7	8/8	8/8
SERV8	6/6	6/6
SERV9	11/11	11/11
computer	11/16 = .69	11/11
Service-Description	15/15	15/15
CD	4/16 = .25	4/4
CD9900SI	3/3	3/3
CD9600SI	3/3	3/3
DVD	3/16 = .1875	3/3
DVD3001I	3/3	3/3
Memory	2/2	2/2

4.2.3. Case 3: Discovering Alternatives for a Generic Service Request

This section presents a short example to illustrate the use of semantic queries to discover Gaia services. This work was reported in [150, 201, 203]. This test shows how the semantic query could be used to discover services in a Ubiquitous Computing Environment.

Assume a user enters a smart room, and wishes to display a presentation on one or more devices. Gaia offers several services that can display information on different displays, e.g., from a PowerPoint file, from HTML or other text document, from JPEG or MPEG graphics, and so on. These services can be classified into related categories, as in Figure 70. To accomplish the user's goal, the Gaia infrastructure needs to compose a service that sends the user's presentation to the displays. The required service depends on the input file(s), the devices and services currently configured, and the user's request.

The application can use a semantic query to discover the classes of service that can display the presentation. For example, the application might request "display this PPT file on a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

wall screen". A semantic query can be formulated requesting an application of class "Display PowerPoint". The query would define the desired concept (essentially, a sub-concept of "PowerPoint Application"), and request all concepts that are equivalent to this concept.

The query is implemented in two parts. The target concept is described using DAML+OIL, as in Figure 108. This DAML+OIL XML defines a dummy class (**Q1**) which is an instance of **PowerPoint**. The query would request concepts that match the concept (using the real name of the ontology):

"http://gaia.net/ontos/ActiveSpace.daml#Q1".

The Ontology Service returns a list of classes based on the current configuration, including not only "PowerPoint", but also "Single PPT" (which displays a single file), "Multi PPT" (which displays multiple files on synchronized displays), "Acrobat Reader", "Slide Show App", and other general classes. Figure 109 shows the query and the matches.

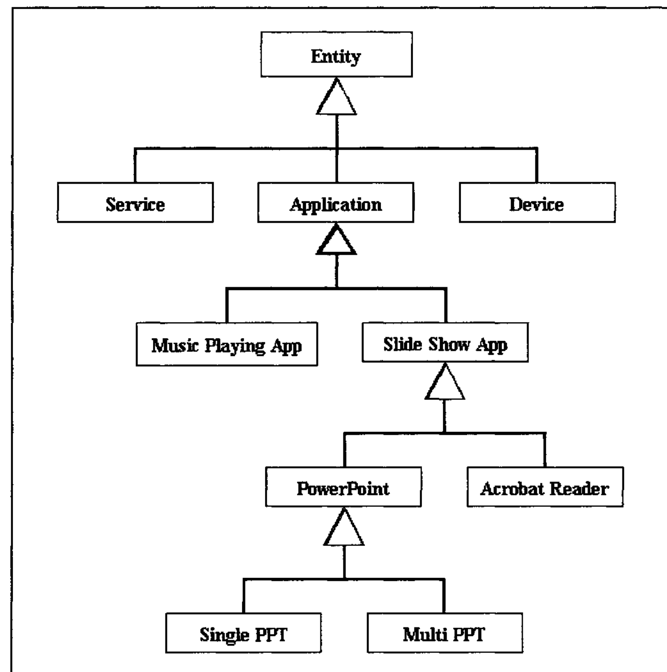


Figure 107. Part of the entity hierarchy of a smart space, with several categories of Application.

For the application, this result means that any instances of these classes are capable of presenting a slide show, perhaps via a format translation (e.g., from PPT to Acrobat). The application (or an agent) uses this list of classes to query the Space Repository to find available instances of these services. The application can sort the classes according to the context of the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

request, e.g., the capabilities of the displays, the number of people present in the room, or other context. If needed, descriptions of the classes can be retrieved from the Ontology Service.

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <rdf:RDF xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
3      xmlns:dc="http://purl.org/dc/elements/1.1/"
4      xmlns:OilEd="http://img.cs.man.ac.uk/oil/OilEd#"
5      xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
6      xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
7      xmlns:xsd="http://www.w3.org/2000/10/XMLSchema#">
8      <daml:Ontology rdf:about="">
9      </daml:Ontology>
10     </daml:Class>
11     <rdf:Description rdf:about="http://gaia.net/ontos/ActiveSpace.daml#Q1">
12         <rdf:type>
13   <daml:Class rdf:about="http://gaia.net/ontos/ActiveSpace.daml#PowerPoint"/>
14         </rdf:type>
15     </rdf:Description>
16 </rdf:RDF>
```

Figure 108. Example DAML description of an abstract class (some details omitted for space).

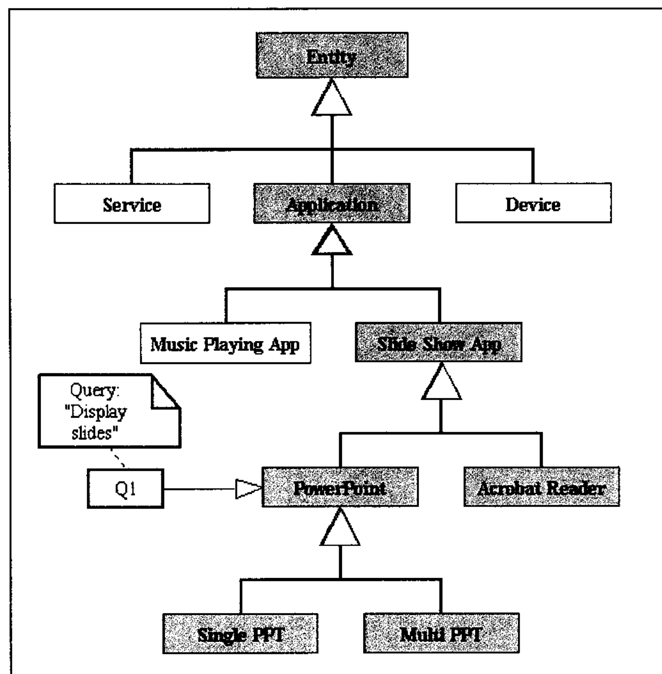


Figure 109. Sketch of the classes that match "Q1".

This example illustrates some of the features of the semantic query. First, the query discovers an interesting set of "related" classes. In contrast, the Gaia Space Repository has little

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

or no information about the relationship of the entities, and therefore cannot determine these related classes.

Furthermore, this result provides information that would be difficult to obtain from the devices or services themselves. In this example, the request to display "PowerPoint" yields the non-obvious result of "Acrobat Reader". This result should indicate that a translation service exists which can display PPT as PDF. Without the Ontology Service, it would be difficult for either the PowerPoint or the Acrobat services to know that this relationship exists in a given space.

4.2.4. Case 4: The "People, Places, Things" Ontology

Several experiments used a version of the People, Places, Things (PPT), Device, and Library ontologies described in Chapter 5, composed as described in Chapter 4. As defined in the earlier Chapter, the ontologies have parallel hierarchies for *objects* and *descriptions* of objects. Semantic queries can be used for any of these classes, but usually the semantic queries would use the *descriptions* because the descriptions have the information of interest to the query (i.e., the attributes of the class of objects).

In order to make the queries and results easier to understand, the experiments reported in this chapter used only the *description* classes from the ontologies. For clarity, the names are simplified here. For instance, the queries used **PersonDescription**, **PlaceDescription**, and **ThingDescription**, rather than **Person**, **Place**, and **Thing**. But in this Chapter, the queries and results are labeled **Person**, **Place**, and **Thing**. This renaming does not affect the algorithm or the results.

The PPT ontology (descriptions only) has 46 classes. Example queries were created, and the results evaluated. Figure 110 and Figure 111 show parts of the class hierarchy of this ontology. The properties of each class and other logical constraints are omitted from the diagrams for simplicity.

Figure 110 shows the logical classes of Input and Output capabilities, including categories for **HardCopyInput**, **HardCopyOutput**, and **HardCopyIO** (both **HardCopyInput** and **HardCopyOutput**). These classes implement the important special case discussed in Chapter 3. Several classes of device are shown, such as **Printer** and **Fax**.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Figure 111 shows logical classes of resources in the library, such as **Book** and **Journal.** These and other classifications are composed to create the overall system ontology, as discussed in Chapter 4.

The ontology also includes a set of Personal Roles for the library. This classification was described in Chapter 4 and developed in detail in other work [148, 149]. The roles were defined in a DAML+OIL ontology, which was added to the composite ontology.

In addition, axioms were added to the ontology to define relationships across the separate sub-ontologies, as discussed in Chapter 5. Figure 112 illustrates an example of such links: **Manual** (as in, "online documentation") is defined to be equivalent to **Web Resource, Web Address** is equivalent to **URL,** and **Author** is defined to be a subclass of **Person.**

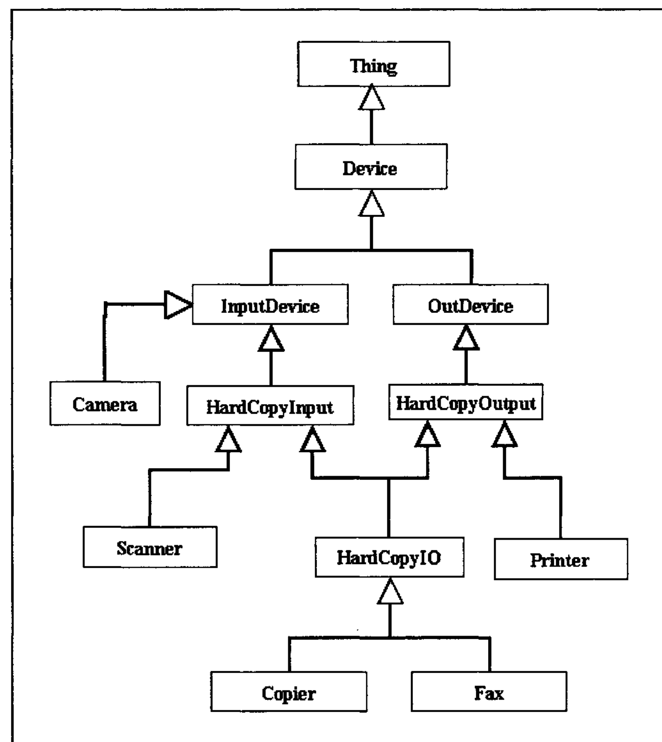


Figure 110. Fragment of the classes of the Devices ontology.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

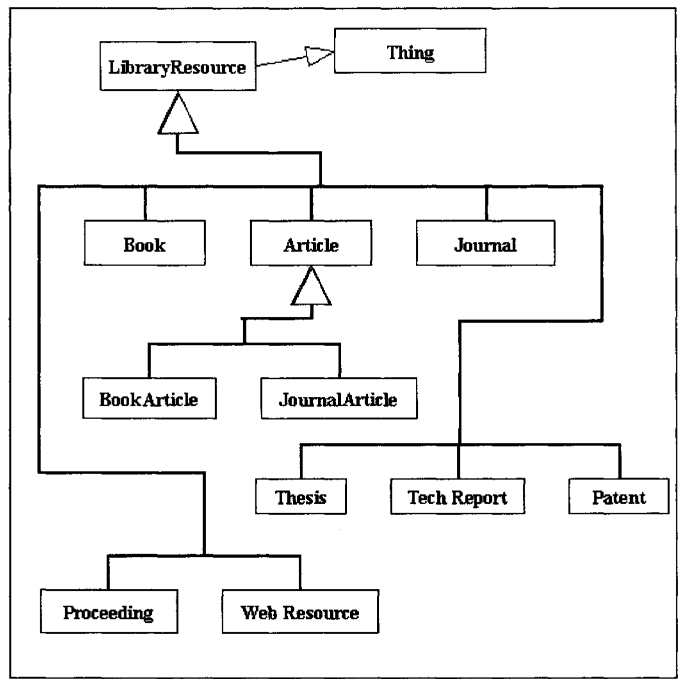


Figure 111. Fragment of the classes of the Library Resources ontology.

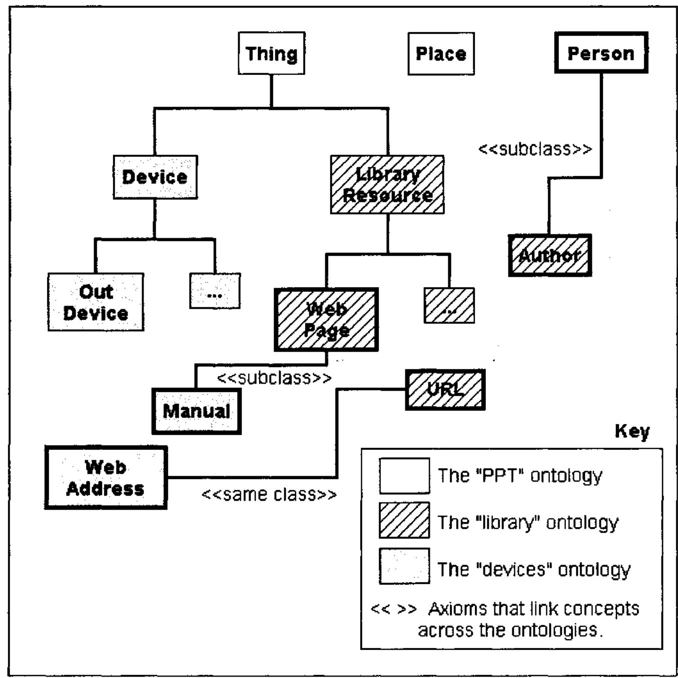


Figure 112. Illustration of some of the links between concepts in the composite ontology.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Example queries were constructed for 46 classes of the composed ontology. Each query was one class in the ontology. Using the composite ontology, the results typically include classes from more than one of the original ontologies, and may well match classes from ontologies other than the source of the query class.

Table 38 and Table 39 list the 46 queries and the recall and precision statistics for each query. The recall was perfect: every result included all the possible correct answers. For 12 queries, the result included at least one false positive.

Table 38. Recall and Precision statistics for 46 queries (1 of 2). (Note: these results include the query as a match for itself, i.e., every query must match at least one class.)

Query	Recall	Precision	
Camera	1	9 / 9	1.00
Copier	1	9 / 9	1.00
Device	1	11 / 11	1.00
Fax	1	9 / 9	1.00
HardCopyIO	1	9 / 9	1.00
HardCopyInput	1	8 / 9	0.89
HardCopyOutput	1	8 / 9	0.89
InputDevice	1	9 /12	0.67
Manual	1	4 / 4	1.00
OutDevice	1	8 / 12	0.75
Printer	1	8 / 8	1.00
Scanner	1	8 / 8	1.00
Article	1	5 / 6	0.83
Author	1	3 / 3	1.00
Book	1	3 / 3	1.00
BookArticle	1	5 / 5	1.00
Creator	1	4 / 4	1.00
ISBN	1	1 / 1	1.00
ISSN	1	1 / 1	1.00
Journal	1	3 / 6	0.50
JournalArticle	1	5 / 5	1.00
LibraryResource	1	13 / 13	1.00
Organization	1	2 / 2	1.00
Patent	1	3 / 3	1.00
Proceeding	1	4 / 4	1.00
TechReport	1	3 / 3	1.00
Thesis	1	3 / 3	1.00
WebPage	1	4 / 4	1.00
Person	1	2 / 2	1.00
Place	1	1 / 1	1.00
Thing	1	24 / 24	1.00

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 39. Recall and Precision statistics for 46 queries (2 of 2).

Query	Recall	Precision	
Role	1	13 / 13	1.00
Graduate	1	4 / 5	0.80
Professor	1	4 / 5	0.80
Role#Staff	1	9 / 9	1.00
Student	1	5 / 8	0.63
TA	1	4 / 5	0.80
UnderGrad	1	4 / 5	0.80
UniversityRole	1	13 / 13	1.00
Clerk	1	4 / 5	0.80
Librarian	1	5 / 5	1.00
LibraryRole	1	13 / 13	1.00
Patron	1	3 / 3	1.00
Staff	1	5 / 5	1.00
WebAddress	1	2 / 2	1.00
URL	1	2 / 2	1.00

Table 40 shows more detail for some of the queries with correct results. The queries correctly discover classes from several of the composed ontologies. Notably, concepts that were defined to be equivalent using axioms are correctly discovered. For example, 'devices.daml#Manual' matches 'library.daml#WebPage', and 'library.daml#Author' matches 'PPT.daml#Person', as implied by the DAML axioms.

The 12 queries that have less than perfect precision (i.e., one or more false positives) reveal possible weaknesses in the ontologies and/or query algorithm.

The queries for the hard copy devices (e.g., **Printer**) give correct results as defined in Chapter 3 (see Table 40). However, the query for **InputDevice** and **OutDevice** returns false positives (Table 41). The results include both input and output devices for both queries. Figure 113 illustrates the match for **InputDevice**, and Figure 114 shows the match for **OutDevice**.

This result appears to reflect a weakness in the design of the ontology: the hard copy output is correctly defined, but the higher-level classes do not define the concept of an Input-output device. Under the ontology used, any sub-class of **HardCopyIO** is a sub-class of **InputDevice** and also **OutDevice**. The ontology could be changed to add a new classification, **IODevice**, which is a sub-class of (**InputDevice** AND **OutDevice**), similar to the definition of **HardCopyIO**.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Queries for two of the sub-classes of the **LibraryResource** had low precision. Table 42 shows results for two classes that had false positives. The classes **Article** and **Journal** match each other. It is not clear why this match occurs.

Table 40. Sample queries and results (match to self omitted for space).

Query Class	Matches (self omitted)	Precision	Recall
library.daml#JournalArticle	library.daml#BookArticle library.daml#LibraryResource library.daml#Article PPT.daml#Thing	4/4	4/4
library.daml#BookArticle	library.daml#JournalArticle library.daml#LibraryResource library.daml#Article PPT.daml#Thing	4/4	4/4
devices.daml#Camera	devices.daml#Scanner devices.daml#Copier devices.daml#Device devices.daml#Fax devices.daml#InputDevice devices.daml#HardCoyInput devices.daml#HardCopyIO PPT.daml#Thing	9/9	9/9
devices.daml#Scanner	devices.daml#Copier devices.daml#Device devices.daml#InputDevice devices.daml#HardCopyInput devices.daml#HardCopyIO PPT.daml#Thing	7/7	7/7
Devices.daml#Printer	devices.daml#Device devices.daml#OutDevice devices.daml#HardCopyOutput PPT.daml#Thing devices.daml#Copier defices.daml#Fax devices.daml#HardCopyIO	8 /8	8 / 8
library.daml#WebPage	devices.daml#Manual library.daml#LibraryResource PPT.daml#Thing	4/4	4/4

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 41. Questionable Results for classes of Device

Query Class	Matches (self omitted)	Precision	Recall
InputDevice	devices.daml#Device devices.daml#OutDevice (false positive) devices.daml#Copier devices.daml#HardCopyIO devices.daml#Scanner devices.daml#Camera devices.daml#Fax devices.daml#HardCopyInput PPT.daml#Thing devices.daml#Printer (false positive) devices.daml#HardCopyOutput (false positive)	9/12 = .75	8/8
OutDevice	devices.daml#Device devices.daml#InputDevice (false positive) devices.daml#Copier devices.daml#Fax devices.daml#HardCopyIO devices.daml#Printer devices.daml#HardCopyOutput PPT.daml#Thing devices.daml#Camera (false positive ?) devices.daml#HardCopyInput (false positive) devices.daml#Scanner (false positive ?)	8/12 = .67	8 / 8

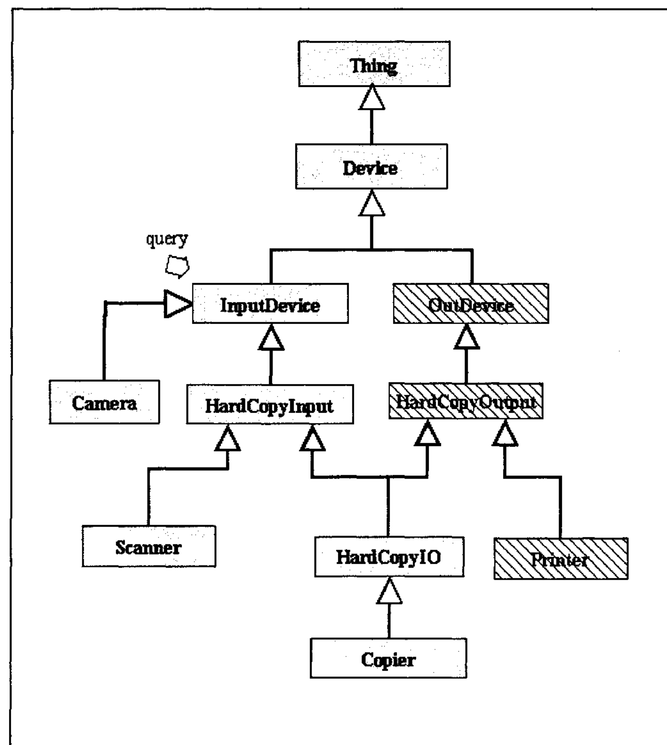


Figure 113. Sketch of the classes that match a query for InputDevice (see Table 41).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

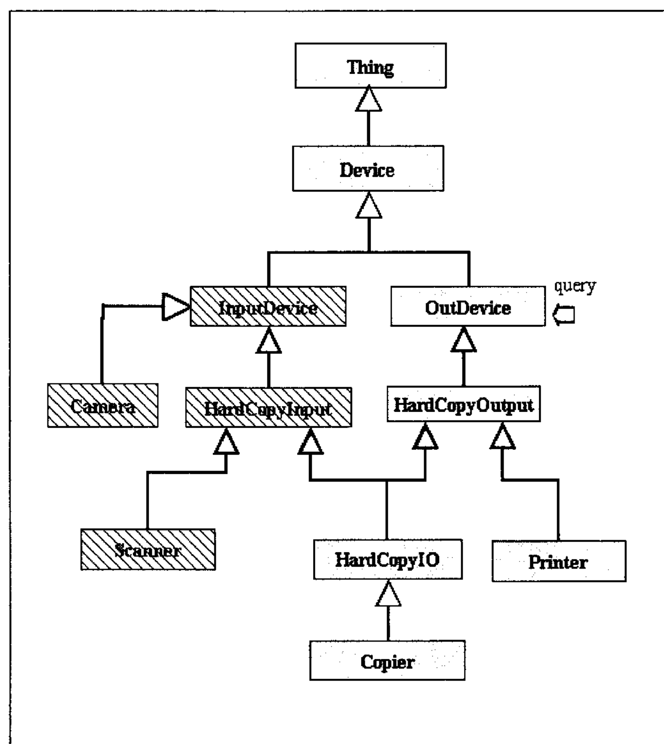


Figure 114. Sketch of the classes that match a query for OutDevice (see Table 41).

Table 42. Questionable results for library resources.

Query Class	Matches (self omitted)	Precision	Recall
Article	library.daml#BookArticle library.daml#JournalArticle library.daml#LibraryResource PPT.daml#Thing library.daml#Journal (false positive?)	5/6 = .83	5 / 5
Journal	library.daml#LibraryResource PPT.daml#Thing library.daml#Article (false positive?) library.daml#BookArticle (false positive?) library.daml#JournalArticle (false positive?)	3/6 = .5	3 / 3

Several of the queries for Roles had comparatively poor precision. In some cases, this is due to judgment calls: **Graduate** is defined as an incorrect match for **UnderGraduate**, but this might be a good match for some users. Similarly, **TA** could be a good match for **Professor** in some cases, e.g., when searching for teachers.

Other cases indicate that the Role ontology is flawed or conceptually confused. First, the Roles have no properties, they are just a classification. Second, the classification is not well

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

designed. For example, the Role ontology defines **Student** and **Staff** to be distinct roles, even though the role **TA** is usually filled by a person who is both **Student** and **Staff**. The ontology is confused, so the queries do not give precise answers.

Overall, the results show that the semantic query algorithm usually produces intuitively reasonable and possibly useful result sets for this non-trivial ontology. Examination of the errors show that there are three common sources of errors:

1. The judgment of correctness is not always clear cut, as in the case of some of the Roles.
2. When the concepts (classes) in the ontology have few properties defined, or the properties are unique to a class, then the classes may be indistinguishable and may be reported as a false positive.
3. The matching give accurate results for classes near the leaves of the taxonomy, but the results near the root have poor precision.

Table 43. Questionable results for personal roles

Query Class	Matches (self omitted)	Precision	Recall
Graduate	Roles#UnderGrad (false positive?) Roles#Student Roles#UniversityRole PPT2.daml#Role	4 / 5 = .8	4 / 4
Professor	Roles#TA (false positive?) Roles#UniversityRole Roles#Staff PPT2.daml#Role	4 / 5 = .8	4 / 4
Student	Roles#Staff (false positive) Roles#UnderGrad Roles#Graduate Roles#UniversityRole PPT.daml#Role Roles#Professor (false positive) Roles#TA (false positive?)	5 / 8 = .63	5 / 5
TA	Roles#Professor (false positive?) Roles#UniversityRole Roles#Staff PPT.daml#Role	4 / 5 = .8	4 / 4
UnderGrad	Roles#Graduate (false positive?) Roles#Student Roles#UniversityRole PPT2.daml#Role	4 / 5 = .8	4 / 4
Clerk	library.daml#Librarian (false positive?) library.daml#LibraryRole library.daml#Staff PPT.daml#Role	4 / 5 = .8	4 / 4

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.2.5. Case 5: Scenario for "Semantic Discovery" of Entities

A fourth case study used the composed ontology described above to demonstrate the matchmaking scenario described in the introduction. In the scenario is that the user or user agent seeks to obtain hard copy output in the current space. He may never have been in the space before, and, in any case, the configuration is dynamic. Therefore, the user or user agent does not know what services are or might be available.

The user or agent must:

1. Discover what services are available
2. Select one
3. Connect to the service

The desired query should be simple (e.g., "Print this here"), yet should return a set of reasonable candidates. The system cannot assume that the user or user agent knows the whole ontology. For example, an application should not need to know all classes of devices that might deliver "print" service, nor the current contents of the room.

In the example ontology there are several classes of entities that can provide hard copy output; printers, network attached copiers, fax machines, and so on. This is encapsulated in the ontology as a taxonomy of functions (i.e., service descriptions). For example, Figure 115 shows part of the Devices ontology that defines categories of input and output, along with categories of devices, such as **Printer** and **Copier**. This ontology represents a hypothetical system in which, although these devices may have different interfaces, they are all potentially usable to obtain hard copy output.

The approach is to query the Ontology Service to discover a set of classes that are logically related to the query. Then the user or user agent will search the Space Repository or other services to discover available instances of these classes in the current space. From the second list, the best candidate may be selected, and the service may be invoked.

As the system evolves, the ontology is updated to include new kinds of devices. This is implemented by adding new concepts to the ontology using the composition algorithm defined in Chapter 3. Queries to the Ontology Service will return new values as the ontology changes.

This scenario was demonstrated in a series of queries. The initial ontology is similar to Figure 110 above (composed from the People, Places, and Things (PPT), Library, and Devices

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ontologies), except the class **Fax** has never been defined. This represents the ontology for a particular smart room that has never has a fax machine. Figure 115 shows the salient part of the ontology in the initial state.

The query for 'print this here' is formulated as a description of the requested service. In this example, the query is defined by describing an instance of the concept **Printer** that has specific attributes. Figure 116 shows DAML+OIL XML defining **Printer-001**, which is the target of the query. This concept is added to the ontology (composed), and then the query is executed to fine concepts similar to **Printer-001**. The concepts that match are shown in Figure 117, which is the expected set.

Ignoring the trivial responses (**Thing**, **Device**, etc.), the query gives the interesting set of classes, including **Copier**. The implication of this result is that, if the space contains no printers, but does have a copier machine, it may still be possible to satisfy the query, i.e., the user may be able to "print" by sending the document to the fax or copier. This result correctly reflects the relationship between the services, i.e., they are all subclasses of **HardCopyOutput**.

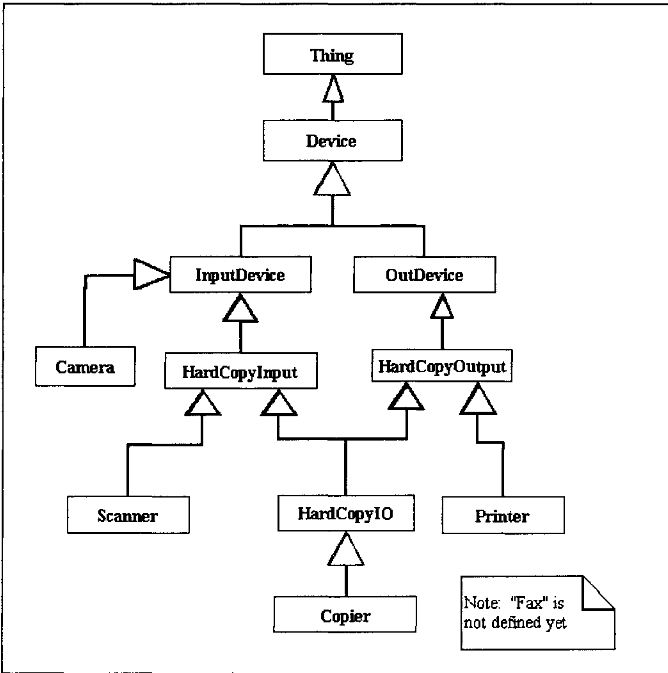


Figure 115. System ontology: "Fax" has not been defined. (Compare to Figure 110.)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
 1 <?xml version="1.0" encoding="ISO-8859-1"?>
 2 <rdf:RDF xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
 3     xmlns:dc="http://purl.org/dc/elements/1.1/"
 4     xmlns:OilEd="http://img.cs.man.ac.uk/oil/OilEd#"
 5     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 6     xmlns:ns0="http://somewhere.net/devices6.daml#"
 7     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
 8     xmlns:xsd="http://www.w3.org/2000/10/XMLSchema#">
 9     <daml:Ontology rdf:about="">
10     </daml:Ontology>
11     <rdf:Description
rdf:about="http://somewhere.net/devices6.daml#Printer-001">
12         <rdf:type>
13             <daml:Class
rdf:about="http://somewhere.net/devices6.daml#Printer"/>
14         </rdf:type>
15         <ns0:DeviceID>
16             <xsd:string xsd:value="prnt-001"/>
17         </ns0:DeviceID>
18     </rdf:Description>
19 </rdf:RDF>
```

Figure 116. Simple Description of the Requested Print Service in DAML+OIL XML.

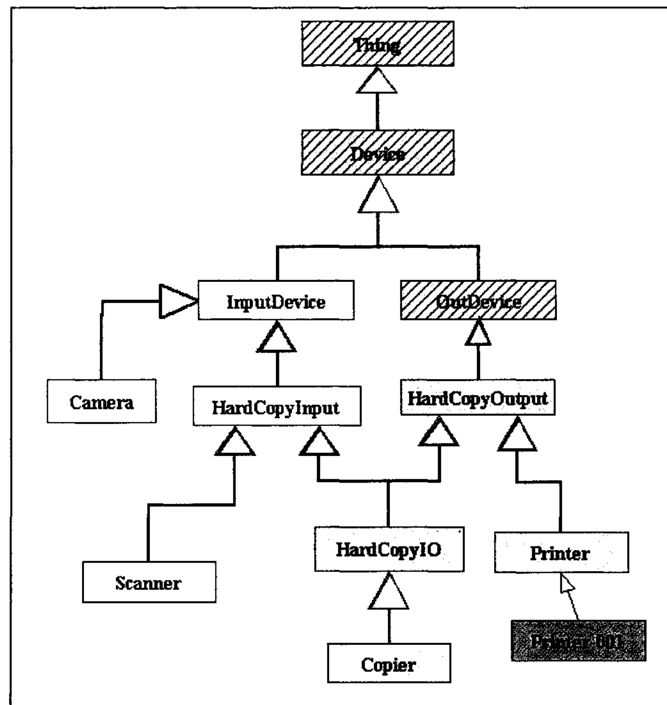


Figure 117. Illustration of the results of a query for "Printer" (trivial results hatched).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

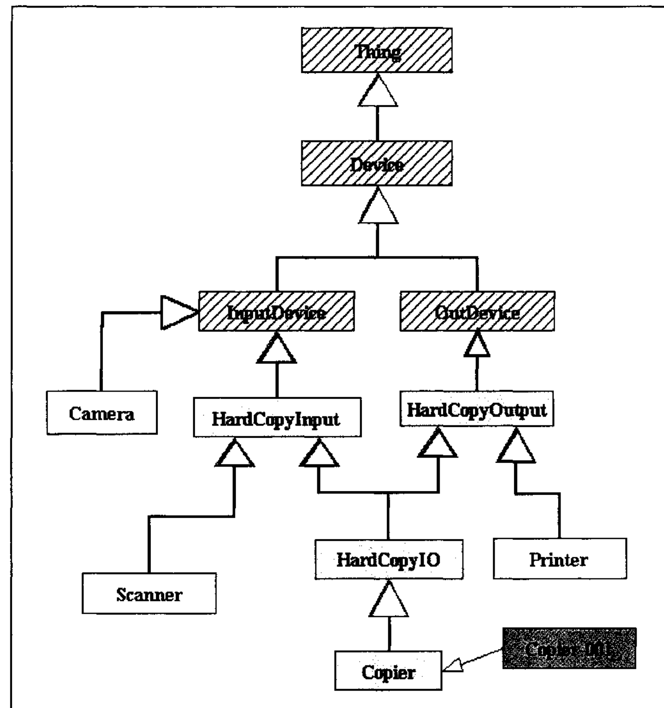


Figure 118. Illustration of the results of a query for "Copier" (Compare to Figure 117).

In the scenario, there is an asymmetrical relationship between the **Printer** and **Copier** service, **Copier** may substitute for **Printer**, but **Printer** does not match **Fax**. In this example, a query that seeks to discover **Copier** devices, gives results something like Figure 118. Ignoring the trivial hits, the **Copier** is matched by **HardCopyIO**, and so on. Note that a request for **Copier** does *not* return **Printer** as a result. This shows that the Ontology Service correctly implements this important case, as discussed in Chapter 3.

The scenario continues: now the system configuration changes. A fax service is installed in the room. At this time, the *room* has the same configuration as Figure 110, but the Ontology Service still has no knowledge about **Fax** devices yet. A query about **Fax** will return empty, and no other query can discover **Fax** devices. For example, the query for **Printer** obviously cannot return **Fax** as one of its results. It is necessary to update Ontology Service, so that queries can discover the new service.

Initially, the **Fax** device can simply be added to system repositories. For example, Figure 119 shows the ontology with a category called **Fax** and an instance of that category. With this composite ontology, the queries illustrated above were repeated. The query for **Printer** will discover the same classes as before, not including the **Fax** (Figure 120). Similarly, a query for

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Fax will discover only itself (Figure 121). The **Fax** device has not (yet) logically associated with **HardCopyIO** or any other classes of the ontology.

The Ontology Service needs to be updated to reflect the relationships of the devices and services. This problem is solved simply and efficiently using the composition algorithm of the Ontology Service. When the **Fax** is added to the system, a DAML+OIL ontology is created to describe the **Fax** concept. Figure 122 shows a simple example of such a DAML file: a class **Fax** is defined, which is a sub-class of **HardCopyIO**. This ontology is composed into the system ontology, as discussed above.

The DAML ontology for **Fax** can contain much more complex information that shown in this example. For instance, the definition of the class can be a full definition, including properties, relations, and restrictions. It can also define new properties. Figure 123 shows a more elaborate definition of the **Fax** class. This DAML file can be used instead of the file shown in Figure 122.

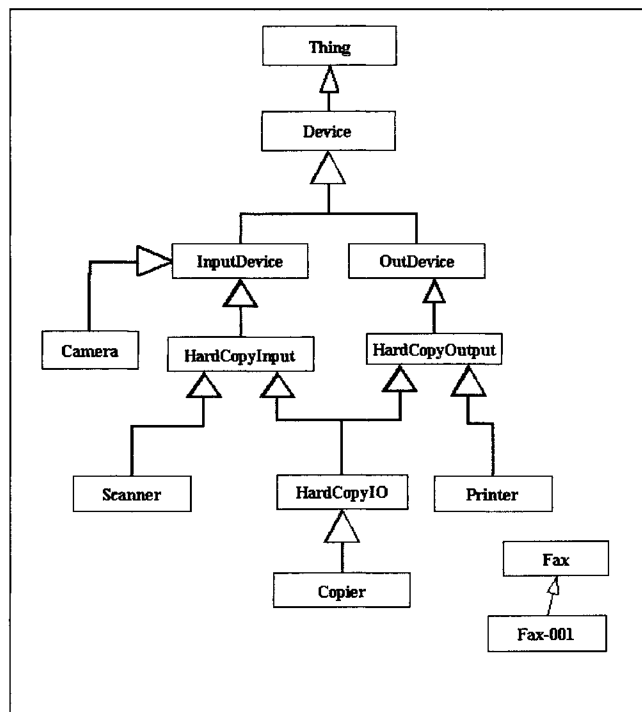


Figure 119. A new device is added, but the class "Fax" has not been connected to the other classes in the KB.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

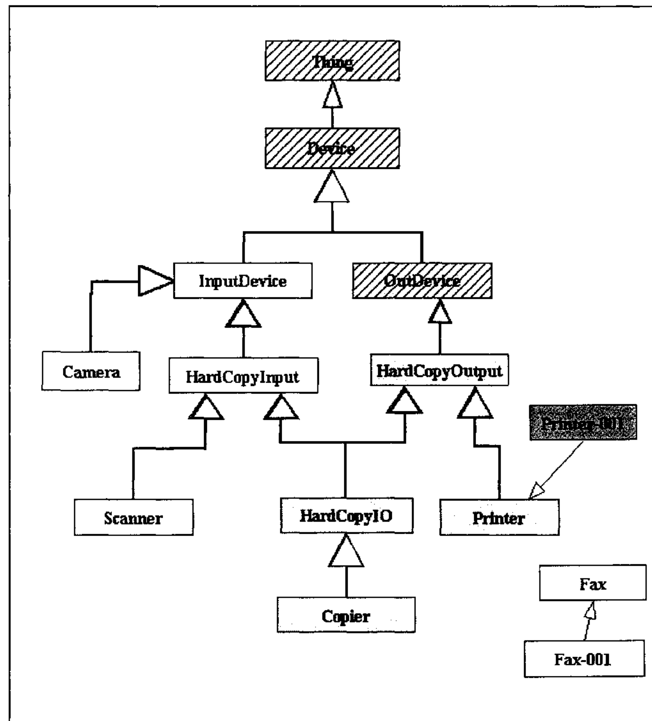


Figure 120. A query for "Printers" does not discover the Fax (compare to Figure 117).

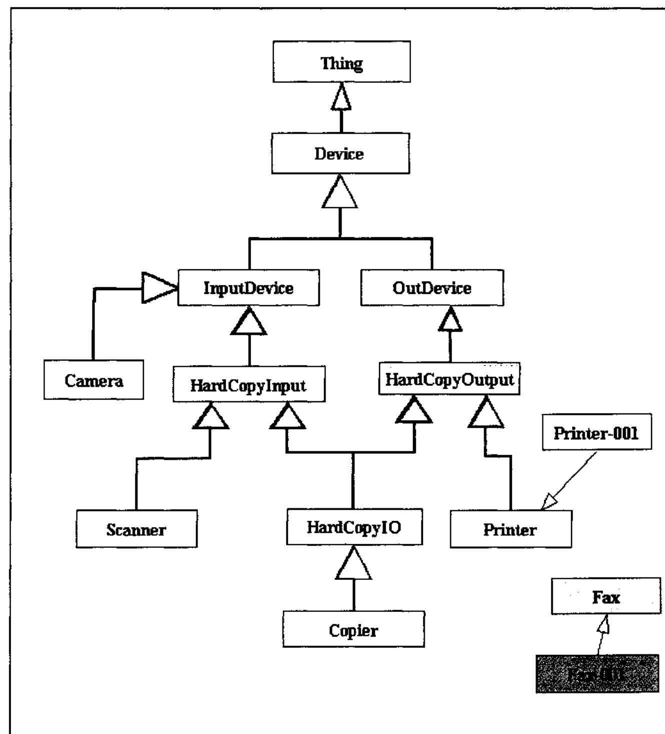


Figure 121. A query for "Fax" discovers only itself (compare to Figure 118 above).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <rdf:RDF xmlns:daml="http:www.daml.org/2001/03daml+oil#"
3     xmlns:dc="http:purl.org/dcelements/1.1/"
4     xmlns:OilEd="http:img.cs.man.ac.uk/oil/OilEd#"
5     xmlns:rdf="http:www.w3.org/1999/02/22-rdf-syntax-ns#"
6     xmlns:rdfs="http:www.w3.org/2000/01/rdf-schema#"
7     xmlns:xsd="http:www.w3.org/2000/10/XMLSchema#">
8     <daml:Ontology rdf:about="">
9     <daml:Ontology>
10    <daml:Class rdf:about="http:somewhere.net/devices.daml#Fax">
11       <rdfs:subClassOf>
12     <daml:Class rdf:about="http:somewhere.net/devices.daml#HardCopyIO">
13        <rdfs:subClassOf>
14    <daml:Class>
15 <rdf:RDF>
```

Figure 122. The minimal definition of the class 'Fax'.

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <rdf:RDF xmlns:daml="http:www.daml.org/2001/03daml+oil#"
3     xmlns:dc="http:purl.org/dcelements/1.1/"
4     xmlns:OilEd="http:img.cs.man.ac.uk/oil/OilEd#"
5     xmlns:rdf="http:www.w3.org/1999/02/22-rdf-syntax-ns#"
6     xmlns:rdfs="http:www.w3.org/2000/01/rdf-schema#"
7     xmlns:xsd="http:www.w3.org/2000/10/XMLSchema#">
8     <daml:Ontology rdf:about="">
9     <daml:Ontology>
10    <daml:Class rdf:about="http:somewhere.net/devices.daml#Fax">
11        <rdfs:subClassOf>
12    <daml:Class rdf:about="http:somewhere.net/devices.daml#HardCopyIO">
13        <rdfs:subClassOf>
14        <rdfs:subClassOf>
15            <daml:Restriction daml:cardinalityQ="1">
16  <daml:onProperty rdf:resource="http:somewhere.net/devices.daml#hasFax">
17<daml:hasClassQ rdf:resource="http:www.w3.org/2000/10/XMLSchema#boolean">
18            <daml:Restriction>
19        <rdfs:subClassOf>
20        <rdfs:subClassOf>
21            <daml:Restriction daml:cardinalityQ="1">
22<daml:onProperty rdf:resource="http:somewhere.net/devices.daml#hasColor">
23<daml:hasClassQ rdf:resource="http:www.w3.org/2000/10/XMLSchema#boolean">
24            <daml:Restriction>
25        <rdfs:subClassOf>
26        <rdfs:subClassOf>
27            <daml:Restriction>
28 <daml:onProperty rdf:resource="http:somewhere.net/devices.daml#Density">
29 <daml:hasClass rdf:resource="http:www.w3.org/2000/10/XMLSchema#integer">
30            <daml:Restriction>
31        <rdfs:subClassOf>
32    <daml:Class>
33<daml:DatatypeProperty rdf:about="http:somewhere.net/devices.daml#hasFax">
34        <rdfs:domain>
35   <daml:Class rdf:about="http:somewhere.net/devices.daml#IODevice">
36        <rdfs:domain>
37        <rdfs:range>
38            <xsd:boolean>
39        <rdfs:range>
40    <daml:DatatypeProperty>
41 <rdf:RDF>
```

Figure 123. A definition of 'Fax' including properties and constraints.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Note that, in order to add the **Fax** to the ontology, the vendor or system administration is required to state that "Fax is-A HardCopyIO". The distributor or installer does not need to know anything else about the current ontology. The **HardCopyIO** is defined in a public ontology, which is available to anyone who needs to register devices.

After the Fax ontology is composed into the current system ontology, the composite ontology is the same as shown in Figure 110 above. When the queries for **Printer** and **Fax** are repeated, they give the answers as in Figure 117 and Figure 118, as intended.

4.2.6. Summary of Queries

The experiments reported in this section show that the queries give correct results within the limits of the information in the ontologies and Knowledge Base. The results reflect the subsumption hierarchy and other constraints defined in the ontologies.

The results of the queries discover "interesting" matches, even when they are defined in different input ontologies. Also, the answers change as the system evolves. These results show that the prototype Ontology Service can answer queries in a dynamic system with concepts from multiple ontologies.

5. Results 2: Performance of the Composition Algorithm

This section presents a series of experiments that measured the performance of the implementation of the composition algorithm in the prototype Ontology Service. The first experiment measured the basic case, loading an empty ontology. The second experiment measured the loading and composition of the ontologies described above. The third experiment investigated the performance of the prototype as the ontology grows larger.

These experiments were performed in the environment described in section 4.1. Clearly, the results would be different for another system, configuration, or alternative Knowledge Base. This study did not attempt to systematically evaluate these variables.

5.1. Basic Overhead: Load the Empty Ontology

The first experiment measured the overhead of processing an empty ontology, which is a baseline for the cost of the communication and other overhead. Table 44 shows the measure times to load and verify a DAML+OIL ontology with no objects. In this case, the time to verify the ontology is simply the time to setup and conclude the verification procedures, i.e., the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

overhead for any verification. Parsing the XML takes about 50 ms minimum, while a call to the verify operation takes at least 500 ms, even for an ontology with no concepts.

Table 44. Measured time to load the empty ontology (ms), average of five trials.

Operation	Time ms	% time
Parse XML	48.2	8
Verify	534.6	90
Other	14.0	2
Total	596.8	100

It is important to note that these times are very sensitive to the system configuration and the versions of the software. This table gives a baseline for the system used for all the measurements in this section. Reductions in this overhead would affect all the times reported below.

5.2. Load and Compose the Person, Places, and Things Ontology

In the second experiment, a system ontology was composed from the examples in Chapter 4, as described in section 5 above. This process has three steps (excluding initialization with the empty ontology):

1. Load the top level People, Places, and Things (PPT) ontology. This is implemented by composing the PPT with the empty ontology.
2. Compose the Library ontology, with links and axioms.
3. Compose the Device ontology with links and axioms.

At each stage, the composition algorithm described in Chapter 4 is executed. The input ontology is validated, the input is merged into the current ontology and then the composed ontology is validated. When additional axioms and links are specified, they are composed with the system ontology, which is then validated again.

Figure 124 and Table 45 show the total time to compose the three ontologies, along with their links and axioms. The total time of the composition depends on the size of the input ontology and the size of the composed ontology. As the current ontology grows, validating the composed ontology dominates the time.

The results in Table 45 clearly indicate that the cumulative size of the system ontology is the main determinant of the performance. The next section measures the performance in more detail.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

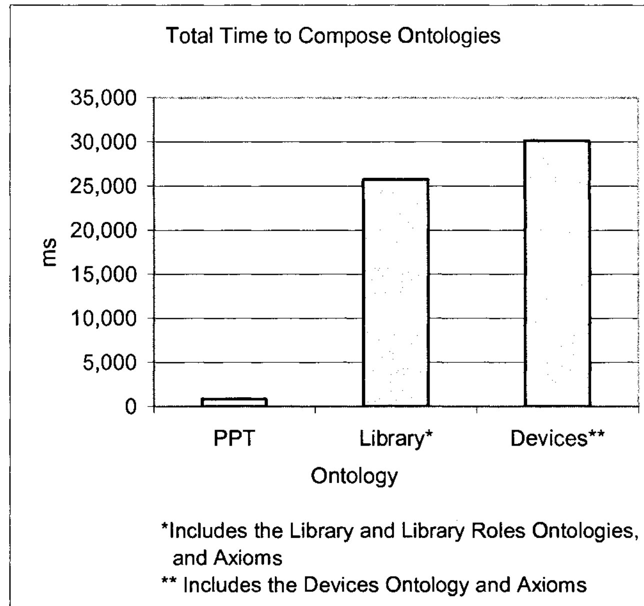


Figure 124. Total time to compose ontologies (see Section 4).

Table 45. Total time to compose three ontologies (average of five trials).

Ontology Added	Input Concepts	Cumulative Concepts	Total time (ms)
PPT	4	4	867
Library	30	34	25,743
Devices	14	48	30,137

5.3. Scale-up: A Larger Ontology

In order to understand the cost of the different steps, a third experiment composed additional ontologies to create a larger system ontology. The goal was to increate the number of concepts without significantly changing other aspects of the ontology. To do this, a dummy ontology was created by making a copy of the Library ontology with the objects renamed. Composing the dummy ontology essentially adds a duplicate of the Library ontology, i.e., each concept and relation in the Library is added to the ontology again. The resulting ontology is structurally similar to the original, except it has more concepts.

First, the ontologies described above were composed, then four dummy ontologies (labeled (A-D) were composed to measure the performance of the server. In total, the experiment merged in eight ontologies (counting the Library Roles ontology), plus linking axioms, for a total of 10 steps, excluding the initialization (the empty ontology). The final size of the ontology was 136 concepts.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Figure 125 shows the total time to compose the ontologies, as the additional ontologies are added. As expected, the run time increases as the ontology grows. Table 46 gives a more detailed view of the total time for the stages of the composition process for the ontologies.

In Table 46, each of the composition steps is listed. For the Library ontology, there are three steps, add the Library (ontology 23 classes), add the Roles ontology (7 classes), and add the axioms. For each stage, the input ontology (or axioms) is parsed and validated, the input is added to the current ontology, and then the composed ontology is validated. Similarly, for the Device ontology, the ontology is added, then axioms. The dummy ontologies have no axioms.

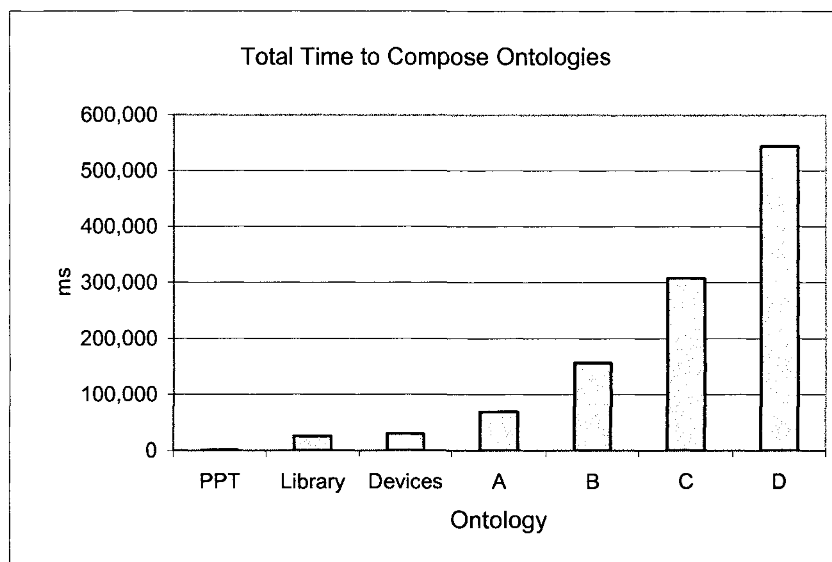


Figure 125. Total time to compose ontologies.

Table 46. Measured time (ms) for composing ontologies, with the size of the ontology at each step. Percentage of total time is shown in parentheses.

Ontology	**PPT**	**Library**			**Devices**		**A**	**B**	**C**	**D**
Cumulative size	4	27	34	34	48	48	70	92	112	136
Load+ verify	712 (82%)	9,549 (83%)	7,581 (83%)	4,148 (81%)	15,569 (88%)	11,132 (91%)	65,069 (94%)	151,706 (97%)	301,017 (98%)	535,652 (98%)
XML	124 (14%)	1,579 (14%)	1,174 (13%)	739 (14%)	1,174 (7%)	811 (7%)	2,994 (4%)	4,300 (3%)	5,025 (2%)	6,119 (1%)
Other	30 (3%)	349 (3%)	372 (4%)	250 (5%)	1,049 (6%)	400 (3%)	1,011 (1%)	1,094 (1%)	1,860 (1%)	2,190 (0%)
Total	867	11,478	9,127	5,137	17,793	12,343	69,075	157,100	307,903	543,961

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

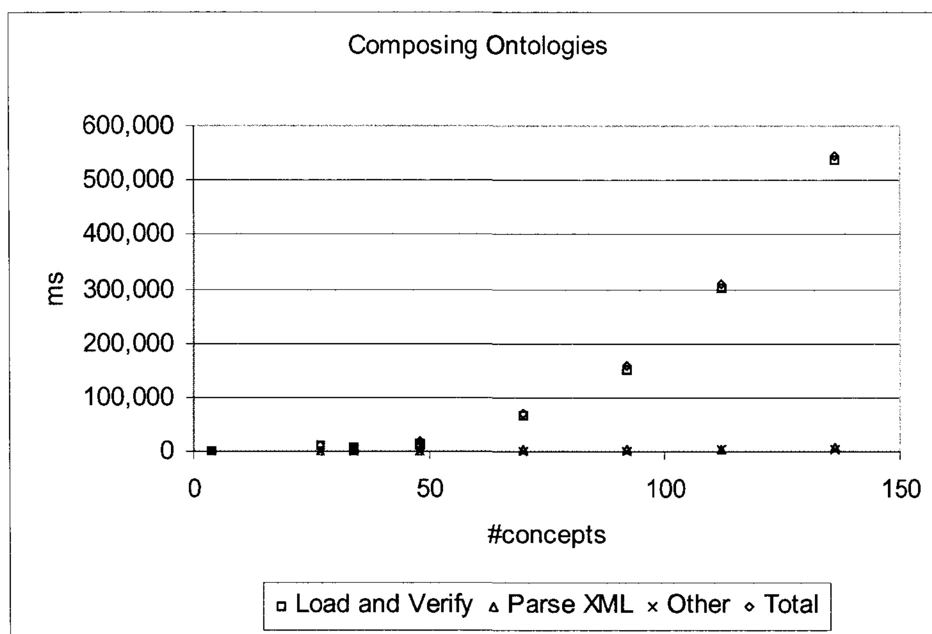


Figure 126. Measured run time (ms).

These data show that the total time is dominated by the 'verify' operation (from 81 to 98% of the total). The XML parsing and overhead is a small fraction of the total, which is approximately proportional to the size of the ontology.

Figure 126 shows a plot of the data from Table 46. This plot suggests clear that the total time to compose ontologies increases with the size of the cumulative ontology, and the total time is dominated by the time to load and validate the ontology, i.e., to load the ontology into the Knowledge Base and then prove that the KB is consistent.

Figure 127 gives a separate plot for the load and verify steps. This figure makes clear that loading the ontology into the FaCT Knowledge Base (i.e., asserting all the concepts) requires time proportional to the square of the size of the ontology.

Figure 128 shows the elapsed times for all the processing except the load of the KB. This graph indicates that all these times are comparatively small, although the verification time (the time to prove each concept is satisfiable) is proportional to the square of the size of the ontology.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

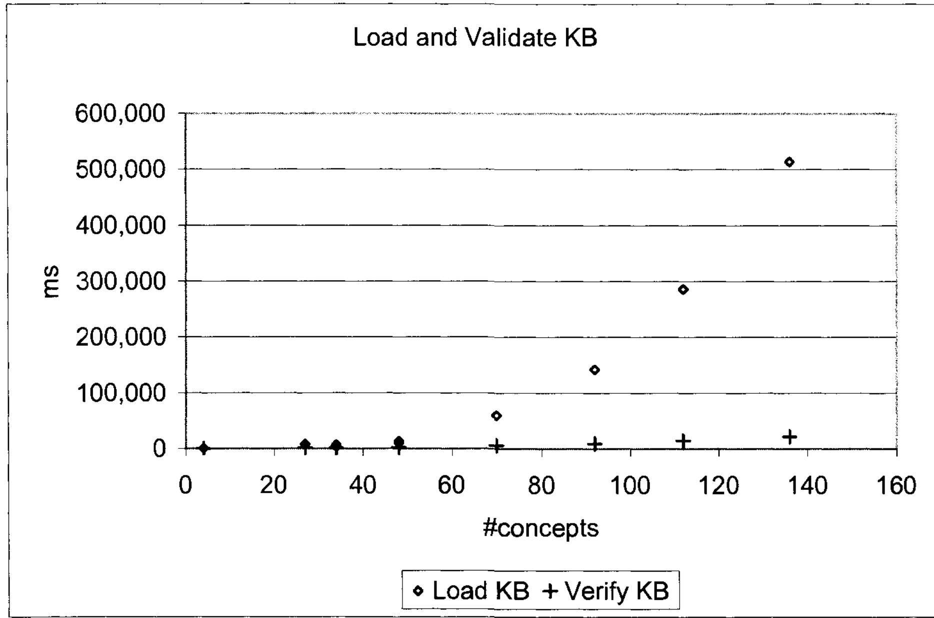


Figure 127. Time to Load and Validate.

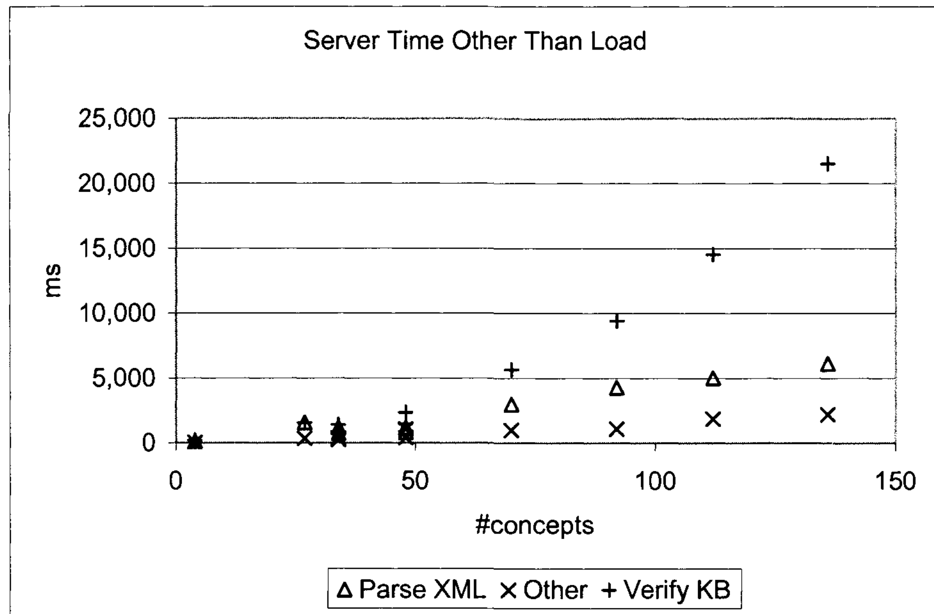


Figure 128. Elapsed time for processing other than loading the KB.

5.4. Summary

These experiments show that the performance of the implementation of the composition algorithm is dominated by the performance of the FaCT server. Specifically, loading and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

verifying the ontology is the majority of the time. In this study, these steps executed in time proportional to the square of the number of concepts in the ontology.

As discussed in Chapter 4, loading and validating the ontology is implemented as a series of calls to the Knowledge Base. Loading the KB is implemented as a series of assertions, proportional to the size of the ontology. Verifying the ontology is implemented as a series of queries (proofs), one for each concept in the ontology. Clearly, these steps will be proportional to the size of the ontology. In this study, the loading step took time proportional to N^2, where N is the number of concepts in the ontology. This indicates that, in the FaCT server, each assertion requires time proportional to the size of the Knowledge Base.

These results indicate that a faster Knowledge Base, e.g., Racer [94-96], could significantly improve the overall performance, at least for some environments.

6. Results 3: Performance of the "Semantic Query" Algorithm

Three experiments were conducted to evaluate the performance of the query algorithm. The first experiment measured the time to complete a query that has no matches, i.e., a miss. The second and third experiments measured the time to execute the queries described in Section 3.

These measurements were done in the same environment as described in section 4.1.

6.1. Overhead

In order to estimate the basic overhead for a query and response, a deliberate "miss" was measured. That is, the "miss" is a completely unknown string, in which case the server processing is minimal. This query is resolved by looking in cached data structures, no calls to the Knowledge Base are needed. Therefore, the total time for this query is almost entirely overhead.

Table 47.The total time (ms) for a query that missed (average over 5 trials).

Total Round Trip	Overhead	Server processing
541.0	363.4	177.6

Table 47 shows the total round trip time from the client. The average elapsed time for a miss was about 540 ms round trip from the client, with approximately 180 ms on the server. The communication overhead was about 360 ms.

6.2. The Service Description Ontology

The second experiment measured the time to complete the queries about the Service Description ontology described in Case 1 above. Each of the 17 concepts in the ontology was

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

used as the query. The results contained 2 to 15 concepts returned as matches. For each query, the system ontology was identical, so the difference in performance was due to the processing required to determine the result.

Table 48 and Figure 129 show the average elapsed time for each query. The results returned for each query were reported in Section 4. The queries required from 350 ms to over 4,000 ms processing by the server. Figure 130 shows a scatter plot of the total time by the number of concepts matched. This plot shows that the variation in the processing time was *not* correlated to the number of hits returned.

The Ontology Service software was instrumented to report the number of nodes (concepts) visited by the query, i.e., the number of nodes considered by the query algorithm. Column 3 of Table 48 shows the number of nodes visited by each query. These results show that the test queries visited 2 to 43 nodes. (A miss would visit zero nodes, a node could be visited more than once if it matches more than one of the criteria.) Figure 131 shows a scatter plot of the total nodes visited by the response time. The total time to process the query is correlated to the number of nodes visited.

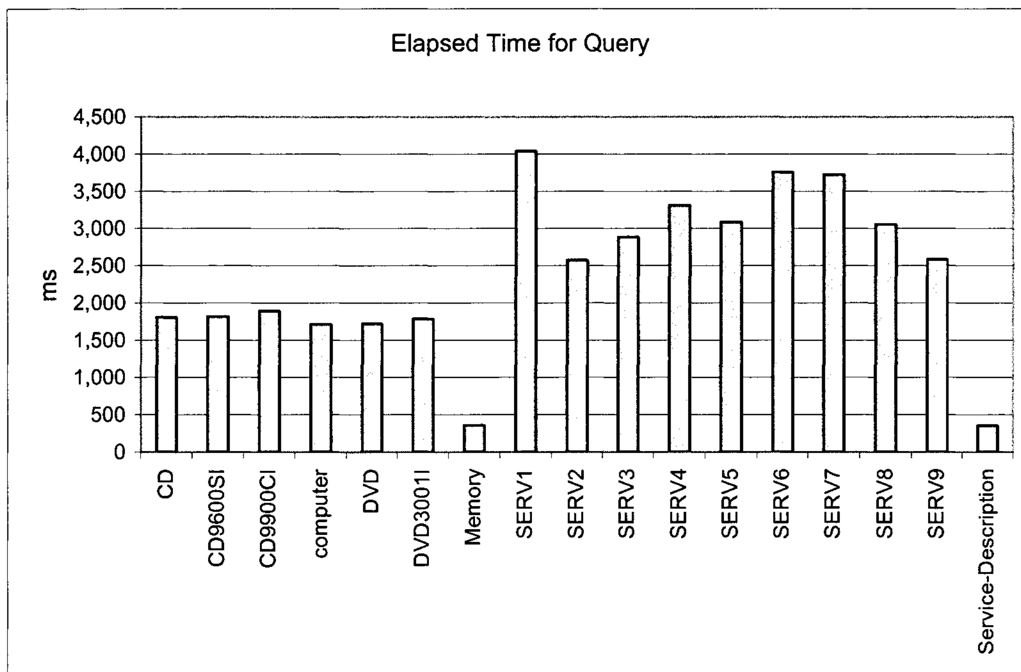


Figure 129. Total time to process queries.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 48. Total hits, nodes visited, and elapsed time for the test queries.

Query	#hits	Nodes visited	Substitutable tests	Processing Time (ms)
CD	4	19	14	1808.4
CD9600SI	3	17	15	1816.6
CD9900CI	3	17	15	1890.8
computer	16	26	14	1714.4
DVD	16	17	14	1720.2
DVD3001I	3	16	14	1786.4
Memory	2	2	0	358.6
SERV1	10	43	34	4039.8
SERV2	11	29	22	2573.8
SERV3	7	33	25	2884.4
SERV4	6	35	28	3308.4
SERV5	8	33	27	3088.8
SERV6	8	42	33	3761.0
SERV7	8	40	32	3725.2
SERV8	6	30	27	3056.6
SERV9	11	34	22	2587.6
Service-Description	15	18	0	354.6

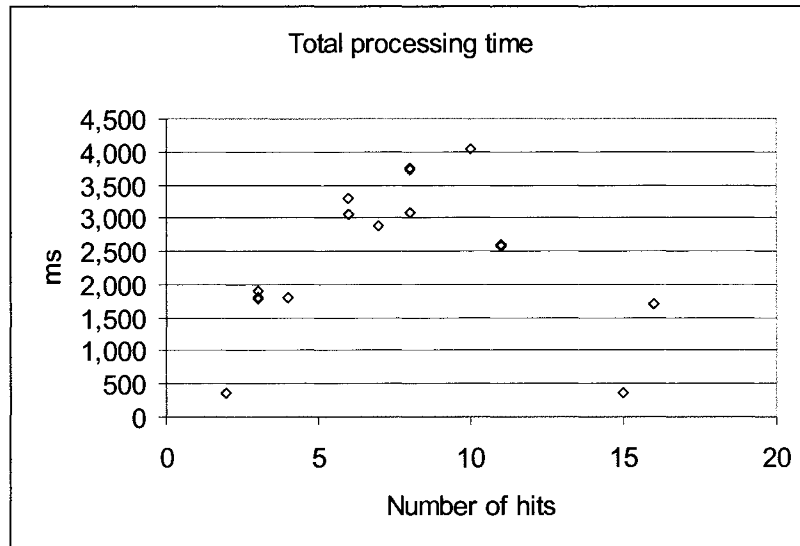


Figure 130. Total processing time by number of hits returned.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

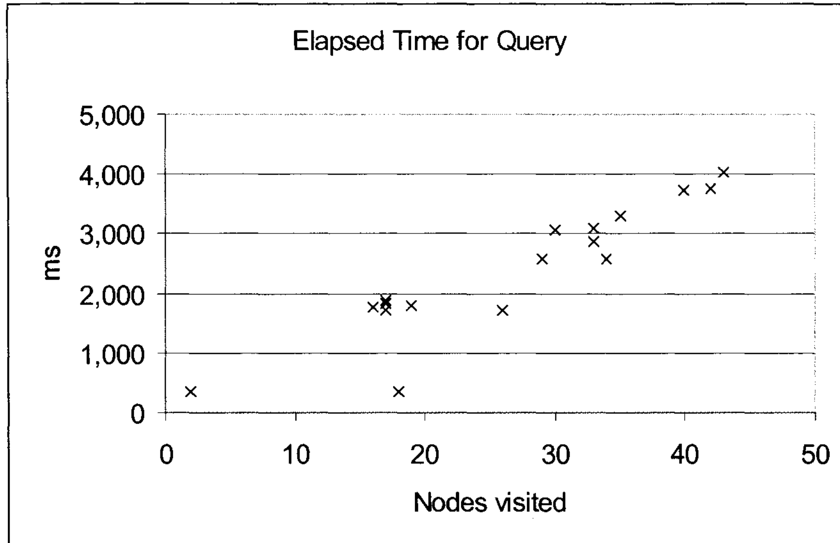


Figure 131. Processing time by total nodes visited by the query.

Recall that while most of the tests defined by the query algorithm can be resolved by reading data structures, the *substitutability* test usually requires either one or two queries to the Knowledge Base (a *subsumption* test for the parents and if that succeeds, a *compatibility* test as defined in Chapter 4). Depending on the ontology and the query, some nodes visited mayor may not require a substitutability test.

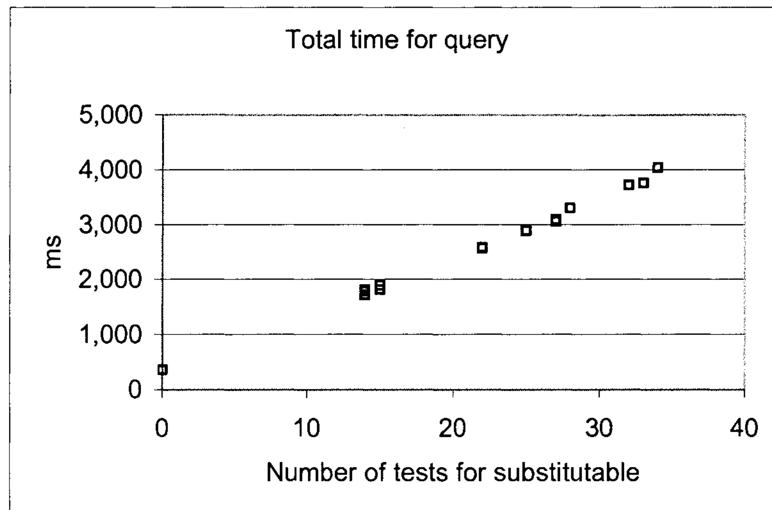


Figure 132. Processing time by the number of substitutability tests..

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Column 3 of Table 48 shows the number of *substitutability* tests done for each query. Figure 132 shows a scatter plot of the number of substitutability tests by the total time. This plot shows that the number of substitutability tests is nearly perfectly correlated with the total time. This results clearly indicates that the number of calls (proofs) to the Knowledge Base is the critical bottleneck.

6.3. The Composed People, Places, and Things (PPT) Ontology

A similar experiment was conducted using the queries to the composed PPT, Library, and Device ontology, described in Section 4 above. After the input ontologies were composed, the system ontology was identical for each query. The sample queries were the 45 concepts defined in the ontologies.

Table 49 and Table 50 show the average elapsed time for each of 45 queries, along with the number of hits and the number of nodes visited. Figure 133 shows a histogram of the average elapsed time for each query. The queries required from 385 ms to 7,633 ms processing by the server. As would be expected, the worst case times are longer compared to the previous experiment.

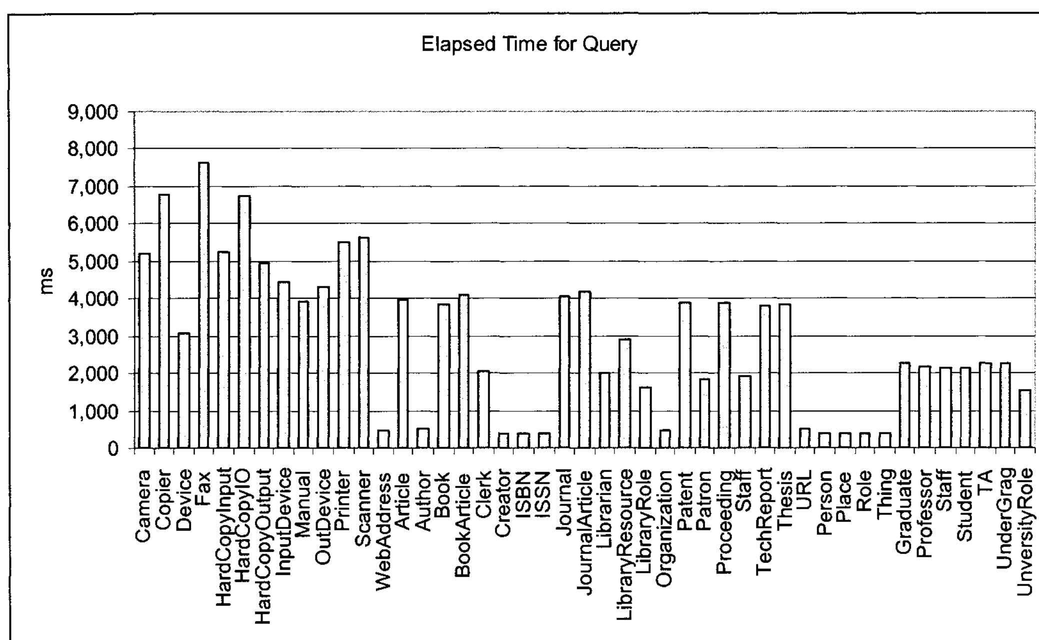


Figure 133. Total time for the queries.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 49. Total hits, nodes visited, and elapsed time for queries (1 of 2).

Query	hits	Nodes Visited	Substitutable tests	Elapsed time (ms)
Camera	9	39	36	5,211.0
Copier	9	54	47	6,771.6
Device	11	35	22	3,070.4
Fax	9	54	47	7,633.0
HardCopyInput	9	46	36	5,267.6
HardCopyIO	9	56	46	6,755.8
HardCopyOutput	9	45	35	4,951.0
InputDevice	12	41	31	4,442.8
Manual	4	35	32	3,943.8
OutDevice	12	39	31	4,324.2
Printer	8	42	38	5,516.2
Scanner	8	43	39	5,646.2
WebAddress	1	5	1	486.8
Article	5	38	32	3,963.6
Author	3	3	1	492.6
Book	3	34	32	3,859.4
BookArticle	5	36	33	4,073.8
Clerk	5	18	15	2,026.8
Creator	4	5	0	380.6
ISBN	1	0	0	385.0
ISSN	1	0	0	384.4
Journal	3	34	32	4,051.8
JournalArticle	5	36	33	4,180.2
Librarian	1	18	15	2,007.0
LibraryResource	13	42	22	2,914.2
LibraryRole	13	18	11	1,610.2
Organization	2	2	1	488.6
Patent	3	34	32	3,895.6
Patron	1	16	14	1,822.8
Proceeding	4	34	32	3,863.6

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 50. Total hits, nodes visited, and elapsed time for queries (2 of 2).

Query	hits	Nodes Visited	Substitutable tests	Elapsed time (ms)
Staff	1	20	14	1,908.6
TechReport	3	34	32	3,785.4
Thesis	3	34	32	3,853.6
URL	1	5	1	510.8
Person	2	3	0	392.8
Place	1	0	0	382.6
Role	13	14	0	382.2
Thing	24	25	0	384.4
Graduate	5	20	17	2,251.4
Professor	5	20	17	2,155.2
Staff	1	22	16	2,115.0
Student	8	22	16	2,139.2
TA	5	20	17	2,257.2
UnderGrad	5	20	17	2,243.0
UnversityRole	13	20	11	1,548.4

Figure 134 shows that, as in the first experiment, the total time for the query is correlated with the number of substitutability tests. This plot is extremely similar to Figure 132, for the queries of the first experiment. The slopes of the lines are similar, approximately 110 ms per substitutability test. These results are consistent with the results from section 6.2.2.

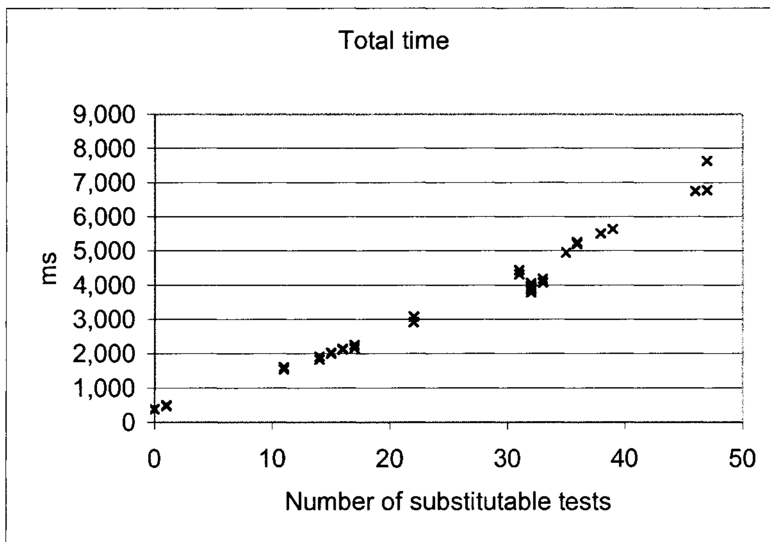


Figure 134. Total time by number of substitutable tests.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

6.4. Summary

These experiments show that the overall time for answering queries is not proportional to the size of the ontology, and it is not proportional to the number of answers returned. The analysis of the queries indicated that the performance is limited by the *substitutability* tests. This result can be explained by the fact that the other types of queries (exact matches, subsumption) can be answered from the data structures, i.e., by traversing the graph in memory. However, a *substitutability* test requires either one or two queries to the Knowledge Base. As in the earlier results, the performance of the Knowledge Base is the limiting factor in the performance of the prototype Ontology Service.

The absolute time to execute a query was much less than the compose operation. In the worst cases, the query takes almost 8 seconds, which is rather slow. Examination of traces from the prototype suggests that many nodes were unnecessarily visited several times. A preliminary examination suggests that eliminating this and other unnecessary work could reduce the total processing time, perhaps by 25-50% for some queries.

The queries used in the measurements did not use a query description, which provides a much more general capability. Recall that when the query has a query description, the processing has three additional steps:

1. Save the current ontology
2. Compose the query description into the current ontology
3. Execute the query
4. Restore the original ontology.

The second and fourth steps each require loading and validating an ontology at least as large as the current ontology. From the results presented here, it is clear that, as the ontology grows, these steps would take far more time than the rest of the query.

The implication is that it will be impractical to exploit the full capability of the query mechanism until the performance of the Knowledge Base is substantially improved.

7. Discussion

This chapter presented an empirical evaluation of the prototype implementation of the Ontology Service. The evaluation showed that the implementation is correct. The prototype

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

performed adequately for small ontologies, but likely would not scale up to larger ontologies, even with a faster system and more memory. The key bottleneck is the Knowledge Base.

7.1. Correctness: Precision and Recall of the Queries

Precision and recall statistics were computed for a set of test queries. The Ontology Service was shown to have very high recall, which means that all possible correct answers will be found. The results had high recall even when the answers came from multiple input ontologies. The query mechanism successfully answered queries that could not be answered from the individual ontologies alone, and successfully discovered implicit relationships that were not explicitly coded into the ontologies. This is a crucial advantage of using the Ontology Service.

The Ontology Service had high precision except for a few cases. This means that the set of answers sometimes included concepts that were rated to be incorrect matches. These errors occurred for concepts that were poorly defined in the ontologies.

In most systems, there is a trade-off between recall and precision. That is, the answer can spread the net wide to catch as many of the desired fish as possible, with the risk that undesired fish will be caught as well. Conversely, the selection can be narrow, which avoids false hits, but may miss some possible correct answers.

The Ontology Service is intended to give a broad set of "similar" classes, especially compared to a registry such as a name service. This goal is met by emphasizing recall, even at the cost of some precision. The analysis presented in this Chapter showed that the prototype has a good balance, with only a few false hits.

The results from the queries are intended to be processed and filtered by a query agent or other system component. Recall that the matches from the Ontology Service are the concepts or classes of entity that might match a request. It will be necessary for the caller or an agent to discover what instances of the classes are currently available. This step may also filter the results according to additional criteria, e.g., preferences.

This filtering step should be able to discard the false hits. The basic idea is that a query agent or other entity calls the Ontology Service in order to discover as many matches as possible. Since the false positives can easily be detected and discarded by the filtering agent, a few false positives are less damaging than missing some possible good matches.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

7.2. Performance of the Service

The experiments reported in this chapter showed that the performance of the composition algorithm was limited by the load and validation steps, which call the Knowledge Base. These steps to time proportional to the size of the ontology (i.e., the number of concepts in the ontology). The performance of queries was dominated by the number of calls to the FaCT server. In particular, loading and validating the ontology takes time proportional to N^2, where N is the number of concepts in the ontology to be validated.

The number of calls to the Knowledge Base depends on the structure of the ontology, i.e., the presence of many "sibling" concepts near the query. This is only indirectly related to the size of the ontology.

In these experiments, the processor is 100% busy during the loading operation. It is clear that the overall performance of the Knowledge Base is limited by the processor speed. The data are insufficient to determine if the processing time is also limited by memory.

The FaCT server is reported to be one of the fastest reasoners when it was developed (e.g., [108]). However, this technology is evolving, and many alternatives exist, such as Racer [94-96], Protégé [171], Java Theorem Prover (JTP) [60], F-OWL [237] and an improved version of FaCT [116, 139, 183]. The modular design of the Ontology Service will make it possible to use better Knowledge Bases when they become available.

While the performance of the prototype is adequate for small or moderate sized ontologies, it is clear that the current implementation could not validate large ontologies in reasonable time for human interactions. The environment used in this study was a relatively small system, perhaps representative of a mid-range device in a Ubiquitous Computing Environment. The N^2 performance of the FaCT server shows that even substantial increases in processor or memory would not, in themselves, yield linear improvements in the performance of the Ontology Service.

Fortunately, the composition operation is required only when the system ontology is updated, which is relatively infrequent. Queries do not need to validate the ontology, and so they run much faster.

In the envisioned environment, the Ontology Service manages the ontology for a local environment, so it should not be necessary to support extremely large ontologies. Therefore, the prototype would be adequate for managing the ontology for a local space.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Ch. 8. Summary and Comparison to Other Work

1. Overview

This thesis adapted and extended previous work from several areas of research to address important problems for a Ubiquitous Computing Environment (UCE). This chapter discusses how this thesis has built on related work in the area of information systems, agents, the World Wide Web, and other Pervasive and Ubiquitous Computing projects. This earlier work was the necessary foundation, from which this thesis developed theoretical and practical techniques needed for UCE.

In this thesis, a Ubiquitous Computing Environment was characterized as a local environment in which a very dynamic and heterogeneous set of activities must be supported. In addition, the UCE must manage the physical environment, as well as software and devices. Third, the UCE should be a seamless: mobile users and objects should be able to enter and leave a local environment without user or administrative intervention

The UCE is also characterized by a locality of use, i.e., a local space needs a dynamic "working set" from a larger universe of components and activities. This key insight recasts the way metadata is used, and led to the developments presented in this thesis.

This thesis considered one of the basic problems for any decentralized system, resource discovery. Resource discovery is a key problem for all decentralized systems, including digital libraries [147, 210], intelligent agent systems [87, 129, 168, 226], and the World Wide Web [14, 58, 159, 170, 262]. In all these applications, there are multiple, independent entities that must exchange messages but do not share a single vocabulary or database schema. This thesis observes that UCEs face a similar challenge, and therefore adopts developments from these applications areas.

In a distributed system, resource discovery is done as part of registration, advertising, look up, and event services. In a UCE, these protocols must operate seamlessly and in real time, and should provide a localized view of the dynamic local environment. In order to operate seamlessly, it is necessary to replace manual tasks with automated algorithms as much as possible.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Ontologies have been developed as a language for managing metadata in distributed information systems. Ontologies and the underlying formal logic are extremely general and powerful, so it is no surprise that they can be applied in UCEs. As discussed in earlier chapters, Description Logic, Knowledge Bases, and the Semantic Web XML languages provide the necessary foundation, which must be extended to work well in the UCE.

Ontologies are used to manage schemas and data in information systems and the World Wide Web. These systems focus on organizing and managing complex bodies of data. The ontologies are created manually; to construct a single, large, slowly changing, schema for the system.

Fusing and matching data from multiple sources is a critical problem for these applications, ant this requires matching across multiple schemas (or, more abstractly, multiple ontologies). The general practice is to conduct a manual cross-walk, analyzing the ontologies to discover trans-ontology relationships. The formal complexity of this problem is not known, but no general solution is known, and some very similar problems are known to be undecidable [17, 74, 98, 193].

In contrast, for a Ubiquitous Computing Environment the central challenge is diversity and real time change, rather than scale and complexity of the information to be managed. Furthermore, the system must operate seamlessly, with little human intervention, so manual cross walks are not feasible. This thesis presented two techniques to address these challenges.

First, the locality principles proposed in this thesis suggest that ontologies for UCE should be developed piecemeal. The ontology languages discussed in this thesis are well suited for encoding many small ontologies. Second, this thesis proposed that each local space should use a dynamically constructed "working set" from a overall pool of ontologies. While familiar from other contexts such as virtual memory systems, this principle has not been used for ontologies before. In order to achieve this, it was necessary to develop a method for automatically composing multiple ontologies with minimal human intervention.

Broker- and agent- based systems use ontologies to improve discovery of that fill certain goals, i.e., given a user request, automatically locate and execute services to meet the goal. This thesis builds on previous work that has developed *matchmaking*, and so called *discovery protocols*. If the UCE is conceived as an environment in which many brokers and agents will

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

operate, it is clear that a common metadata language is needed, along with common services to manage and query using the metadata.

This thesis built on efforts that focused on heuristics to use proofs in Description Logic to define a set of concepts that match a given query concept. The previous work was synthesized and extended to create queries that seem to better meet the queries of a UCE.

Most Pervasive and Ubiquitous Computing Environments have ad hoc metadata, with little or no use of formal languages and reasoning. It is now widely recognized that semantic services will be needed. This thesis has showed the need for ontologies in Ubiquitous Computing, and developed theory and key algorithms to make this possible.

The following sections review this work in more detail, to highlight on the distinctive contributions of this thesis.

2. Ontologies in Information Systems and Knowledge Engineering

Ontologies have been developed and used in Knowledge Engineering and information systems. In these applications, ontologies are used to help build and maintain a large and complex data dictionary, especially across multiple sources, usually as an aid to human analysts and users. The formal taxonomy of an ontology can help develop correct and consistent database schema and abstract class graphs (e.g., [186, 217]), and to improve queries and caching between distributed databases (e.g., [78]).

These applications generally have a human in the loop, both in the construction and use of the system. For example, ontologies are used to aid analysts to develop large complex systems, such as a digital library (e.g., UMLS [154, 166]). The ontologies may also be used in the user environment, to aid the construction of queries (e.g., [218]) or computer aided design (e.g., [158]). These systems focus on organizing and manage complex data for data intensive and knowledge based applications. The systems are relatively static and centralized, so the ontologies can be constructed manually into a single system.

Large scale systems with multiple databases need to combine data from independent sources. In order to do so, it is necessary to align or compose schemas from multiple databases. (Composing two arbitrary ontologies is essentially the same problem as automatically integrating or matching two database schemas.) This problem has several technical challenges:

1. alignment of names

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2. identification relationships between entities the schemas
3. proof that the combined schema is valid

Ontologies have been used to address these challenges. An ontology language provides a uniform namespace, along with mechanisms to prove consistency and other properties. Cross ontology relationships can be represented in the ontology language, as well.

However, cross ontology (cross schema) relationships are difficult to identify. The general practice is to conduct a manual cross-walk, analyzing the ontologies to discover trans-ontology relationships (e.g., [56, 186, 218]). The formal complexity of this problem is not known, but no general solution is known, and some very similar problems are known to be undecidable [17, 74, 98, 193].

This task is aided by tools such as OilEd [8], Protégé [170], or OntoEdit [181, 221] and similar research environments (e.g. [57, 156]). These tools help human analysts visualize the ontologies and can store and validate proposed mappings between ontologies. These tools generally include a Knowledge Base and manage a single merged ontology, which can be exported as OWL XML or other encoding.

Some approaches to schema matching require a cross-walk, but do not require a global, merged ontology. Borgida and Serafini propose a distributed approach, in which multiple separate ontologies are related through "bridge rules" [17]. Others have proposed similar approaches in the domain of multiple database schemes (e.g., [74, 97, 98, 193, 227]).

Automated learning has been applied to attempt to discover trans-ontology relationships between large ontologies. For example, the OntoMerge project promises to merge any two DAML+OIL ontologies using automated learning to discover bridging axioms [52]. Doan et al. use statistical text processing algorithms to attempt to discover trans-ontology relationships [50, 51], and others have attempted similar learning approaches (e.g., [11, 263]). These approaches depend on the existence of training data (e.g., Web pages) which may not be available in a Ubiquitous Computing Environment. Where these techniques can be applied, the cross ontology relationships can be represented in an ontology language, e.g., as OWL axioms.

As discussed earlier, a UCE faces similar challenges when it must use components from many different sources. However, the Ubiquitous Computing Environment is localized, so the central challenge is diversity and real time change, rather than scale and complexity of the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

information to be managed. Furthermore, the system must operate seamlessly, with little human intervention. Interactive tools and statistical analysis do not address these key requirements.

This thesis presented two techniques to address these challenges. First, the locality principles show that ontologies for UCE should be developed piece meal. With smaller, simpler ontologies, it should be easier to identify and state cross ontology relationships. Essentially, by decomposing the overall ontology, the development process should be tractable, so the relationships should be easier to discover and encode. The ontology languages discussed in this thesis were not designed for encoding many small ontologies, but are, in fact, well suited for this use.

Second, this thesis proposed that each local space should use a dynamically constructed "working set" from a overall pool of ontologies. Almost all other systems aim to construct a single omnibus ontology, so this is an unusual way to use ontologies. In order to achieve this vision, it was necessary to develop a method for automatically composing multiple ontologies with minimal human intervention. This thesis showed how this can be done with a simple application of Description Logic, which can be implemented with ontologies encoded in DAML+OIL.

3. Intelligent Agents and Brokers

In distributed systems, brokers and agents act as proxies for users to organize complex services. Brokers are centralized components, while autonomous agents are distributed components, but otherwise they have similar functions. This technology faces many challenges of authentication, delegation, and workflow management (e.g., planning and resource scheduling). For any non-trivial application, complex models of services are needed, including interfaces, behavior, and dependencies (e.g., FIPA [64]). In some systems, ontologies represent the vocabulary of these models (e.g., OWL-S [142]).

Brokers and agents need to discover resources that fill certain goals, i.e., given a user request, automatically locate and execute services to meet the goal. Since the broker or agent is supposed to be autonomous (i.e., not programmed by the user), discovery may require sophisticated reasoning and complex information about the available environment. This thesis builds on previous work on matchmaking, and so called discovery protocols.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In recent years, several so-called *discovery* protocols have been introduced, with the overall goal of making digital networks easier to create and use (e.g., see [144]). "Discovery" is used to refer to a more spontaneous process, in which entities locate and present themselves to other entities. The most important features a discovery protocol are:

- "Spontaneous" discovery and configuration of network devices and service (i.e., not pre-programmed).
- Selection of specific types of service (rather than specific instances of service)
- Low (preferably no) human administrative requirements
- Automatically adaptation to mobile and sporadic availability
- Interoperability across manufacturers and platforms

Some directory services such as LDAP [255] and CORBA Name and Trader Services [175] can be used for service announcement and requests, but do not themselves specify protocols for spontaneous discovery. These services define interfaces, protocols, and languages for advertising and look up, but do not define the contents of the metadata, or mechanisms for defining and sharing data and metadata.

For example, the CORBA Trading Service is a standard interface for a broker, which defines a language for advertising and query. By design, the standard does not define the define the properties of the advertised services or the legal values of properties, i.e., the standard does not define equivalent of a database schema for the contents of the Trader. This task is left to communities, such as the CORBA Domain Task Forces [174].

To address this requirement, metadata standards are emerging for software components (e.g., FIPA [64]) and hardware devices (e.g., Salutation [208]), as well as many other efforts (e.g., [121, 172, 174]). These types of standards are a model for what is needed for all aspects Ubiquitous Computing Environments. The ontologies of the Semantic Web are a natural means to specify, publish, and manage these metadata languages.

A UCE is often implemented by a combination of brokers and agents, as well as less intelligent services such as registries and event channels (e.g., Gaia [205]). When the UCE is conceived as an environment in which many brokers and agents will operate, it is clear that a common metadata language is needed, along with common services to manage and query using the metadata.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The general purpose metadata language and Ontology Service developed in this thesis can be used for many different services, including agents and brokers. Essentially, the Ontology Service takes the burden off the other services, and provides a common mechanism to improve interoperation. This is a logical extension of the agent-based or broker-based approach that should work particularly well in a localized UCE.

As discussed in earlier chapters, one of the common problems for brokers and agents in *matchmaking*; discovering, selecting, and configuring components that match general specifications or goals. This task requires a general framework for queries, which includes:

1. data structures for representing knowledge
2. a query language or languages
3. a model and format for responses (which may be quite complex)
4. an algorithm for matching, i.e., to compute the response for a given query from a given Knowledge Base
5. protocols for populating and updating the Knowledge Base

This study reviewed the use of ontologies and Description Logic to meet these requirements. Description Logic provides formal logic that can be used to create data structures, along with proofs that can be used to answer queries. This thesis used the Semantic Web XML languages as a format for queries and responses. Note that the composition algorithm discussed above is a key innovation for automatically populating and updating a Knowledge Base in a real time system.

Matchmaking is usually approached by designing a single Knowledge Base to manage all the information about the system, both the schema and the instances. There are many examples of such systems, e.g. [2, 19, 34, 35, 44, 57, 58, 139, 143, 154, 159, 184, 189, 218]. In contrast, this thesis proposed a multi part system, separating the schema management (using ontologies) from managing the real time system state. This design trades complexity of the query protocol (it has several steps) against the efficiency of the specialized parts. In particular, since the Knowledge Base may be a performance bottleneck, so it is used to manage the relatively small and slowly changing ontologies. The complex and rapidly changing system state (i.e., the current set of services, the state of displays, the presence of mobile users and devices) to other services, such as the Gaia registry [152, 203].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Given a structured Knowledge Base, queries can be performed using many algorithms. In this thesis, a Knowledge Base is built using Description Logic, which can implement queries as proofs. Alternatively, the relationships can be represented as a graph, and queries can be implemented as algorithms on the graph. (In fact, automated proofs for Description Logic can be formulated as operations on a graph.) This thesis proposed a hybrid algorithm, using simple graph operations when possible, then using more expensive proofs only when needed.

While precise and efficient algorithms can be designed for a particular purpose, it is quite difficult to create a good general purpose algorithm; one that works well for a variety of applications and Knowledge Bases. This thesis built on efforts to define heuristics that use proofs in Description Logic to define a set of concepts that match a given query concept. The previous work was synthesized and extended to create queries that seem to better meet the perceived queries of a UCE. Overall, the developments presented in this thesis are natural extensions and specializations to previous work, which together address the problems of UCE.

4. The Semantic Web and Semantic Web Services

This thesis used the standards of the Semantic Web, which emerged from the need to access resources on the World Wide Web. This is similar to the challenge of integrating multiple databases, except the number of information sources is preposterous, the sources generally have ad hoc (or no) metadata, and there is little consistency. As the Web has evolved toward a service oriented architecture (as Web and Grid Services), the Semantic Web has evolved towards annotating services as well as data [142, 159].

In some sense, a UCE can be viewed as a sub-set of the overall World Wide Web, although the local resources are not necessarily intended to be widely visible. Therefore, it is possible to construct a local UCE using Web or Grid service standards. However, the UCE envisioned in this thesis is a real-time and local view of services, rather than a window into a global pool of services and objects. Thus, many aspects of the Web or Grid are far more general and coarser-grained than is needed for a UCE.

As discussed earlier, the UCE must manage proxies for a variety of devices and physical objects, not just network services. It would be possible to create a Web Service to wrap a physical object or place (e.g., the books and shelves in the library), each of which could be

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

addressed by a unique URL. It is not clear that there is a great advantage to doing this, and this is perhaps too fine grained for Web Services.

In addition to being highly localized, the UCE is dynamic, and must be updated in real time. Given the scale, complexity, and generality of the World Wide Web, it usually cannot provide real time updates (although some services may do so). For this reason, the Semantic Web generally uses relatively large, complex, and static ontologies to search for diverse pool of resources. In contrast, this thesis developed methods for dynamically constructing a local ontology for each local space. Thus, this thesis has focused on the small, dynamic, and local, rather than the large, general, and static. The previous sections have already discussed the implications of these differences, and how the thesis addressed them.

The Ontology Service developed in this thesis was a prototype of the type of infrastructure that will become a standard part of the World Wide Web and other systems. In recent years, several projects have begun to develop similar services in Web and Grid Services [2, 19, 44, 57, 58, 143, 154, 155, 159, 184, 189, 218], the Semantic Grid [34, 35, 43, 75, 139, 213, 224]. These projects have a variety of goals and designs, but all need ontologies and semantic services, as proposed in this thesis.

5. Ontologies in Ubiquitous and Pervasive Computing

Most Pervasive and Ubiquitous Computing environments do not yet use ontologies or formally defined metadata languages. For example, the Gaia system (without the Ontology Service) manages components through several services including the Space Repository, LUA scripts, and interactive user dialogs [23, 205]. While service interfaces are described with the CORBA IDL, other attributes (e.g., the behavior, dependencies, and invocation of a services) are captured in ad hoc metadata and scripts. When a service lookup is performed, the results are presented as lists to in a user dialog. Clearly, these user dialogs and scripts do not provide seamless operation. Reconfiguring the system requires manual changes to the scripts, and the system can be fragile if components change.

This critique is not meant to disparage Gaia: most Pervasive and Ubiquitous Environments have similar limitations (e.g., [47, 65, 70, 138]). Indeed, the Gaia boot service, space repository, and LUA scripts are powerful mechanisms for managing the system [24, 205]. However, adding ontologies and an ontology service opens the way for improved services, as

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

discussed [152, 198, 200, 203]. This thesis has developed key ideas to enable these developments.

Several current research projects are incorporating ontologies into context-aware, Ubiquitous, and Pervasive Computing Environments, to address these issues. For example, the Context Broker Architecture (CoBrA) is an infrastructure for context-aware computing that includes a service to manage a Knowledge Base constructed from OWL XML ontologies [31]. Ontologies classify and define entities important for Pervasive Computing, including devices, users, and events [32]. The Ontology Service is used to implement an enhanced look up service (termed "context-sensitive resource discovery) [27]. This design is similar to the work presented in this thesis, although it does not recognize the importance of locality, and the queries are less sophisticated than those developed in this thesis.

In another example, in the STEER environment ontologies provide common abstract descriptions of Web Services, which can be used by scripts to dynamically bind to appropriate services in the local environment [143]. Several other projects have proposed similar uses of ontologies. Ontologies may be useful for flexible Human Computer Interfaces (e.g., [38, 230]). In this application, ontologies can help select "semantically" appropriate interfaces (e.g., [5, 223]), and (potentially) could be used to explain the interface to users (e.g., [157]). These projects illustrate the variety of ways that ontologies and an ontology service can be used to enhance the services of a UCE.

The research community is beginning to develop mid-level ontologies for important aspects of Ubiquitous, Pervasive, and Context-Aware computing. The emerging consensus is that ontologies are needed for People and Places (as proposed in this thesis), and also Time, Policies (e.g., security and privacy), Quality of Service, and other domains (e.g., [32]). Together, this work represents an emerging consensus of the importance of ontologies and ontology services for Pervasive and Ubiquitous Computing.

6. Prototypes for an Ontology Service

This thesis presented a design for an Ontology Service, with an open interface to manage ontologies for a local space. This service provides a standard interface and services that can be used by all the components of the space. This service wraps standard libraries and stand-alone Knowledge Bases.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Most systems that manage ontologies are designed as part of interactive environments, with an integrated Knowledge Base, rather than a local service. For example, OilEd [8], Protégé [170] or OntoEdit [181, 221] provide environments for editing and checking ontologies, using one or more Knowledge Base, rather than multiple copies.

The Ontology Service proposed in this thesis extracts the common services used by all these systems, and provides a service interface for managing ontologies. This enables all components to use the same ontologies and Knowledge Base.

The Ontology Service developed in this thesis was an early working prototype, and was published as part of the Gaia source code [152, 200, 203]. Several projects have proposed a similar Ontology Service. The Context Broker Architecture (CoBrA) includes a service to manage a Knowledge Base constructed from OWL XML Ontologies [31]. The "Semantic Grid" needs a similar service, as proposed by [34, 35]. These projects indicate an emerging consensus on the need for such a service, although prototypes are not been widely available.

7. Summary

This thesis builds on earlier work from several areas. Together, this work shows the emerging recognition of the need for ontologies and ontology services in UCE. This thesis presented a systematic development of the theory and key algorithms needed to enable these uses.

The UCE is a dynamic but localized environment. This thesis showed how to use powerful and general techniques from databases, artificial intelligence and the World Wide Web to address challenges for a UCE.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Ch. 9. Conclusion

1. Summary of Thesis

This section briefly recapitulates the major points of this thesis.

1. Consideration of the foundations of Ubiquitous Computing Environments
2. A theoretical model of metadata, and review of logics and languages for metadata and Knowledge Representation.
3. Application of this model to key problems of UCE
4. A methodology for developing ontologies, and several example ontologies
5. An algorithm for semi-automatically composing ontologies
6. A synthesis and extension of algorithms for "semantic matching"
7. A prototype Ontology Service
8. Evaluation of the prototype.

This thesis considered the foundations of ubiquitous computing. A Ubiquitous Computing Environment (UCE) is a complex linking of real world and digital objects. Sensors, actuators, and interactive interfaces implement connections between real world objects and events and digital data that can be used by the UCE. This data can only be used by implementing computer models of the entities of the environment.

This thesis focused on one of the critical challenges for Ubiquitous Computing: a flexible model for metadata. Many kinds of objects, including physical objects, places, and people, must be represented in the Ubiquitous Computing system by machine-readable data, e.g., statements that describe entities and relationships in the computing environment. Because this data is secondary (in that it is *about* the objects of interest), it is usually termed "metadata". The metadata provides a level of indirection between the physical world and the computational models.

In a Ubiquitous Computing Environment, there are many types of system state maintained by system services, such as registries, object repositories, and policy databases. The metadata is an intermediate representation used to exchange information between services, applications, and users. The metadata also provides the needed level of indirection between diverse components of the environment. To implement this indirection, mappings must be

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

defined between three domains: the real world (problems of interest), machine-readable metadata, and formal models. This thesis presented solutions to important parts of the management of shared metadata using ontologies.

The overwhelming challenge to this goal is the diversity of the system. First, the Ubiquitous Computing Environment is heterogeneous, with many kinds of spaces, many kinds of objects, and many different uses. If the system is truly ubiquitous, users, devices, and software must be able to operate "seamlessly" in many specific environments for different tasks.

Second, the system faces a "Tower of Babel" problem. Since the components and environments are developed autonomously, and move freely between environments, it is difficult to assure that they share enough common semantics to work together. This problem is found in many variations in Ubiquitous Computing Environments, as well as many other distributed systems. The conceptual models and technology presented in this thesis can be the basis for "semantic translation" to begin to address this challenge.

This thesis showed how methods developed for intelligent agents and the World Wide Web can be applied to the information exchanges in the Ubiquitous Computing Environment. In particular, the *ontologies* developed in the field of Knowledge Representation can be used to address fundamental problems of Ubiquitous Computing, and to develop a "semantic infrastructure" for Ubiquitous Computing Environments.

Chapter 2 addressed the question, "what should be modeled?" in the Ubiquitous Computing Environment. A conceptual model was developed for the real world objects in a Ubiquitous Computing Environment. The model was developed by asking the classic questions "Who, What, Where, Why, When?" The answers to these questions was stated in terms of People (Who), Places (Where), Things (What), Time (When) and Motivation (Why). In this thesis, the model covers only the static concepts: the base abstractions of the model are "People, Places, and Things" (PPT). The basic model can be extended and specialized as needed. The PPT model was applied to three application environments, Shopping, Health Care, and Library. For each environment, the three basic concepts were expanded to define concepts for the specific environment.

Chapter 3 presented some theoretical foundations. A general model for metadata was presented, along with a key design pattern, a proxy for physical object. *Description Logic* was introduced as a language for metadata. Description Logic is a subset of First Order Logic, that is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

designed to represent and reason about concepts. Chapter 4 shows how Description Logic can be used to implement important algorithms for managing and using ontologies, including an algorithm for composition of two ontologies, and semantic queries.

Combining multiple ontologies in a dynamic, heterogeneous system is a very difficult problem. Composing two ontologies is similar to fusing two database schemas, for which there is no known solution. This thesis developed a composition algorithm that takes two valid ontologies and creates a third ontology that correctly represents the concepts in the input ontologies, plus the trans-ontology relationships, if any. This algorithm is a limited, but practical solution for this important challenge.

This goal can always be accomplished by a cross-walk of the two ontologies, to manually create the third ontology. This manual effort is very labor intensive, and no general, automated algorithm to implement this procedure.

A hybrid architecture for queries was presented, in which a "semantic query" was decomposed into three phases:

1. Discovery of all the classes that match the query
2. Discovery of all the instances of those classes
3. Filtering the instances to match the exact query

The first step is the "semantic query", which was developed in Chapter 4.

The essence of the semantic query is a definition of *conceptual match*, i.e., concepts that are related according to the information in the ontologies and the Knowledge Base. Chapter 4 presented definition of a "semantic match," which is a synthesis and extension of earlier work by Gonzalez-Castillo, et al. [75], Paolucci et al. [174] and Li and Horrocks [131]. The semantic match defines a set of concepts that match the query, using the relationships and constraints defined in the current ontology. This set can be used by services to discover concepts similar to, more general than, and more specific than the query, and to discover synonyms and bridge between local terminologies.

This thesis developed a test of *compatibility* using heuristics to construct a query that will discover clashes between the definitions in the ontology. The compatibility test is not derived from the formal semantics of Description Logic, it is designed to reflect intuitive definitions of "substitutability". The heuristics define how to construct an artificial concept that can be tested

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

using Description Logic. If the query concept is *satisfiable*, the two concepts are logically compatible according Description Logic and the facts in the Knowledge Base.

Chapter 3 presented a methodology for developing ontologies. The foundation of the method is to decompose the problem, to create a hierarchy of ontologies. Each ontology represents concepts for a limited domain. Concepts are related to concepts in other domains as needed. This decomposition principle reduces the effort to tractable pieces, while the composition of ontologies enables the sharing and reuse of the pieces.

The process of creating an ontology has two important phases, knowledge acquisition, followed by encoding in a formal language. The knowledge acquisition process is subjective and labor intensive. Once an abstract model is defined, tools assist in the encoding the ontology in a formal language. In this thesis, the DAML+OIL XML language was used.

The methodology was applied to create an example hierarchy of ontologies. Chapter 5 used the Person, Place, and Thing (PPT) model to create the top-level ontology for this project. The PPT concepts are encoded in DAML+OIL XML. The concepts of the PPT are shown to be instances of classes of two *upper ontologies*, the Basic Formal Ontology (BFO) [15] and the IEEE Standard Upper Merged Ontology (SUMO) [121, 167]. This mapping showed that, in principle, all the ontologies developed in this study are logically related to any ontology based on either of these upper ontologies.

Two domain ontologies were developed, one for Library Resources (e.g., books, articles, etc.), and the other for devices that might be used in a Ubiquitous Computing Environment, such as printers. Each ontology was developed using the OilEd tool [8] and encoded in DAML+OIL XML. Certain classes in the domain ontologies were identified to be related to classes in the other ontologies. These trans-ontology relations were applied when the ontologies were composed into the system ontology.

Chapter 6 described a prototype implementation of an *Ontology Service* that may be used by any entity of the system. The Ontology Service maintains a system ontology and Knowledge Base for the terminology about the software, hardware, environment, and physical entities of the Ubiquitous Computing Environment.

The implementation was built on top of existing software and standards. The Ontology Service can interface to many different Knowledge Bases. The CORBA FaCT server was used as a logic engine and Knowledge Base [9, 10]. The Java package *uk.ac.man.cs.img.oil* [8, 178] was

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

used to manage DAML+OIL XML and to interface with the CORBA FaCT server. Subsequently, it was demonstrated that the Racer logic engine ([94]) can be seamlessly used instead of the CORBA FaCT Server.

Measurements showed that the performance of the composition operation is limited by the speed of the verification when using the CORBA FaCT server. The verification requires $O(N^2)$ time, where N is the number of concepts in the ontology to be verified. The performance is adequate for small or moderate size ontologies, such as would be required for a local space.

The query algorithm was evaluated using the ontologies defined in Chapter 5 and loaded in the Ontology Service. Precision and recall statistics were computed for sample queries. The Ontology Service had very high recall, and high precision in most cases: a query usually returned all classes that match the request, with the risk of a few false positives.

The results show that the implementation correctly implements the algorithm, and gives reasonable and interesting answers to queries. The current implementation of the Ontology Service was clearly biased toward high recall (getting many correct matches), with some loss of precision (occasional false positives). This bias reflected a fundamental design goal: the semantic query is intended to find as many hits as possible. The false positives are acceptable, because they will be mitigated by the filtering the results of the query.

2. Implications

This study supports the notion that the ontologies and XML languages of the Semantic Web are a necessary but not sufficient foundation for semantic services. The Ontology Service successfully built on top of implementations of the Semantic Web standards, implementing critical services to manage and query the ontologies.

The results of this study have implications for the design of future Ubiquitous Computing Environments. First, the Ontology Service (or equivalent service) is a mechanism that can be used to improve the interoperability for a Ubiquitous Computing Environment. Second, the Ontology Service is a prototype and example for future standards for infrastructure. And third, the ontologies are a foundation on which other models and languages will be built.

2.1. Addressing the Semantic Interoperability

Applications and services can use ontologies and the Ontology Service (or equivalent) to implement improved and more intelligent services. With properly designed ontologies, the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Ontology Service can improve semantic interoperability for Ubiquitous Computing Environments.

The ontologies provide a standard language for exchanging information and provide m a layer of abstraction as discussed in Chapter 3. Software components can be designed to manipulate the abstract classes defined in the metadata, rather than the specific objects and entities themselves. As the system runs, the abstract classes must be bound to a specific implementation. In this binding process, objects, events, and software from heterogeneous sources must be "interpreted" within a specific model. In general, the binding process must discover components that are "substitutable" for a particular class, and then construct a "translation" of the interface if needed.

The ontologies encode the information that can be used to implement discovery and translation between terms. As discussed in Chapter 4, the semantic match defines a set of related classes that are similar to the target, and therefore potentially substitutable. The ontology tells which classes are synonyms, either explicitly declared or deduced from the Knowledge Base, and also classes that are "partly equivalent": more specific or more general interfaces than the target, and classes that *substitute* the target, according to the Knowledge Base.

Even if a particular class cannot be used through its class-specific interface, it may be possible to use it through a generic super-class. From the system ontology, the application can discover the nearest enclosing class for two classes, i.e., a common ancestor in the taxonomy. The ancestor class is a "semantic bridge" between the descendant classes, i.e., a class that may provide a generic interface that can fulfill the required operation. In this case, even if the application cannot use the object directly, it may be able to cast it to an appropriate super class.

Returning to the example introduced in Chapter 1, recall that the "smart mug" needs to discover local services and products that are *substitutable* for the generic goal, e.g., a cup of coffee. Let's see how the Ontology Service helps achieve this.

First, consider a local ontology about coffee service and coffee products. The ontology will be initialized with some basic and general concepts, Service, Beverage Service, Customer, Beverage, and so on. When a service is introduced to the environment, it will register with the Ubiquitous Computing Environment. The registration will include advertisements for services and products provided, such as Hot Beverage Service, with products such as "Large Black

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Coffee". These service descriptions will refer to one or more ontologies, which define relations such as "Large Black Coffee" is a kind of "Coffee" which is a kind of "Hot Beverage".

The local registry will use the Ontology Service to load the relevant ontologies from their URLs as needed. This operation must try to automatically will be form a local ontology, which represents set of services and products available in the local space, along with relationships among the categories. *This critical step is enabled by the composition algorithm developed in this thesis.*

The smart mug cannot know exactly what service will be available in a local space, so it needs to query the local space with a generic query, and receive detailed information about the services available. A semantic query for a concept such as "Black Coffee" can be used to discover a set of related concepts *according to the current local ontology*, i.e., the current time and place of the query.

This information from the ontology can enable the smart mug to discover:

- Specific names for products, such as "Small Coffee", "Medium Coffee", and "large Coffee", which are all kinds of "Coffee"
- Similar products, such as "Tea" or "Cocoa"
- Generic classes of products, such as "Hot Beverage"

These facts can be used to formulate better queries (e.g., in order to find a more complete listing, ask for "Hot Beverage"), and to give the user better answers (e.g., match up the user's customary preferences with the locally available products). This intelligence is not possible without the ontologies, Knowledge Base, and semantic query.

It is important to note that the ontologies and semantic queries themselves do not make the components and services compatible or interoperable. The ontologies are a language through which interoperable components can advertise and discover capabilities. The Knowledge Base and the semantic matching integrate statements from many sources, and also deduce relations not explicitly declared. But if components are not compatible, an ontology cannot make them compatible.

Furthermore, the principle of "Garbage In, Garbage Out" applies. The Ontology Service is only as good as the ontologies it is given. If the ontologies do not accurately describe the real world, the results will be invalid for the actual environment. If the ontologies are incomplete or incorrect, then the Ontology Service provides dubious output.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.2. Standards for Semantic Infrastructure

The Ontology Service is a prototype of the type of infrastructure that will become a standard part of future Ubiquitous Computing Environments. In recent years, several projects have begun to develop similar services in Web and Grid Services [2, 19, 44, 57, 58, 154, 155, 159, 184, 189, 218], the Semantic Grid [34, 35, 43, 75, 139], and Pervasive Computing Environments [27, 29-32, 38, 229, 230]. In the future, this work will converge to define common standards in several areas.

Besides a common language such as OWL, there is a need for good top-level and mid-level ontologies for many aspects of Ubiquitous Computing. This thesis has presented a rationale and methodology for designing ontologies piecemeal, and composing them into larger ontologies. This thesis provides insight into the models that are needed, a recommended method for developing ontologies, and a demonstration of how to manage and compose multiple ontologies.

The research community is following this approach to develop mid-level ontologies for important aspects of Ubiquitous, Pervasive, and Context-Aware computing. The emerging consensus is that ontologies are needed for People and Places (as proposed in this thesis), and also Time, Policies (e.g., security and privacy), Quality of Service, and other domains [28].

There are three important areas where standard interfaces should be developed. First, there needs to be a standard interface for accessing Knowledge Bases. Second, there need to be standard services for managing and manipulating ontologies. And third, semantic services should be added to standards for infrastructure protocols, e.g., for object registries and event services.

In the prototype service, the **OntoKB** class provided a generic interface to a Knowledge Base. The Open Knowledge Base Connectivity (OKBC) standard has a similar purpose [26]. The OKBC standard includes important features that the **OntoKB** lacks, especially for managing multiple Knowledge Bases. As the OKBC evolves (in particular, it needs to be updated to work with Semantic Web standards), it may replace the **OntoKB**.

The design of semantic queries requires substantial research and development. The DAML Query Language (DQL) abstract specification gives a useful survey of the challenges posed by design of queries and answers using DAML+OIL ontologies and Knowledge Bases (KB) [62]. The DQL specification is a very complex interface and protocol. Queries may be complex hypotheses with many variables, the results can be large sets from multiple Knowledge

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Bases, and a result may be a complex set of values bound to variables. Furthermore, the results require a provenance (e.g., which KB produced the result) and explanation (why the result is said to be true). The design must be flexible enough to include the necessary services, yet simple enough to be implemented and used in automated protocols.

The overall purpose of the Ontology Service is to support augmented protocols for Ubiquitous Computing Environments, including service registration and discovery, event registration and delivery, and other critical services. Many standard services of the UCE can be augmented to include automated registration of metadata using ontologies and the Ontology Service (or equivalent).

For example, an augmented registration protocol was sketched earlier. The registration protocol would have the following steps:

1. When the service registers with the environment (e.g., the Gaia Space Repository [205]), in addition to the standard information (name, address, software interfaces, etc.), the service specifies one or more ontologies (URLs of DAML+OIL (or OWL) XML files) that are needed to describe the service.
2. *If the ontologies are unknown to the system, the registry downloads them (from a URL) and updates the system ontology using the composition algorithm.*
3. The service describes itself by sending a DAML+OIL (or OWL) file to the registry, referring to terms in the ontologies specified in Step 1.
4. The registry registers the service as an instance of the class described, and extracts additional metadata from the service description and/or the Ontology Service.

An analogous approach can be used in other protocols in the infrastructure, such as an event service (e.g., [200, 205]). When an entity registers with the Event Service to provide (or receive) events, it describes the events of interest using terms from one or more ontologies. Similarly, when an entity receives notification of an event, it can use the semantic query to discover the definition of events, and to discover related events, such as more general categories of the event it received. The information in the ontology service can help autonomous entities interpret the events they receive.

As applications and infrastructure develop semantic services, other common services and interfaces may be recognized. For example, it is likely that many components of the system will

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

want to retrieve and navigate taxonomies and descriptions from the system ontology. It may be useful to develop a standard "Ontology Iterator", for instance.

2.3. Integration of Models and Languages

The prototype Ontology Service manages ontologies that are essentially taxonomies. This study suggests that this service is efficient and useful. However, the DAML+OIL and OWL-DL languages are not sufficient for many important concepts that are needed in the metadata of Ubiquitous Computing Environments. As noted earlier, quantitative concepts are not easy to express in a classification system, and many statements of logic, such as rules, cannot be expressed easily or at all with Description Logic.

In addition to the limitations of a specific ontology language, ontologies (classification) is only one part of the metadata for a Ubiquitous Computing Environment. The environment will be composed of many computational models with other metadata languages, e.g., for planning, policies, and quality of service. As these additional languages and models emerge, they must be integrated together to manage a Ubiquitous Computing Environment. Ontologies can be used to define a common vocabulary for this integration.

Researchers are developing new languages that extend DAML+OIL and OWL to other domains. These languages include DAML-Time [41], RuleML [87], Rei (a language for defining policies) [124], and DAML Query Language [62]. Other similar languages could be integrated with OWL ontologies as well, such as the HQML for quality of service [89, 258].

For example, RuleML is intended to be a standard language for stating logical predicates [87]. In a Ubiquitous Computing Environment there are many uses for such predicates, including definition of constraints, policies, and triggers for actions. In the RuleML, OWL ontologies will serve to define the namespaces for the rules: the rules will have subjects and objects, which will be defined as classes from the system ontology.

Other higher level languages should use ontologies in a similar role. For example, the OWL-S (originally DAML-S) project has defined a language for describing service interfaces and capabilities [2]. The OWL-S language is designed to be used for automatic service composition of Web Services, e.g., by autonomous agents [261]. The language explicitly builds on OWL, to define a vocabulary for service descriptions. OWL-S is an example of how ontologies can be used to develop new languages.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Of course, integration of multiple languages is not trivial. The formal semantics of the ontologies must be mapped to the logic of the other models. For example, in RuleML, the language will define what objects can fill the slots in a predicate. At the same time, the ontology defines what classes are related, i.e., which classes are synonymous or substitutable. If RuleML is to use classes from an ontology in its slots, the two languages must be harmonized, to produce a logically consistent framework.

3. Conclusion

This ideas presented in this thesis may lead to a better understanding of the models that are needed for Ubiquitous computing and how to encode the models in a general and flexible metadata language. The thesis presented a rationale and methodology for designing ontologies piecemeal, and composing them into larger ontologies. The research community is following this approach to develop mid-level ontologies for important aspects of Ubiquitous, Pervasive, and Context-Aware computing.

This thesis showed that the ontologies and XML languages of the Semantic Web are a necessary but not sufficient foundation for semantic services. The Ontology Service is an initial prototype for future infrastructure services, but there are many open issues that need to be investigated.

With properly designed ontologies, the services of the Ontology Service can solve part of the Tower of Babel problem for Ubiquitous Computing Environments.

This thesis adapted and extended previous work from several areas of research to address important problems for a Ubiquitous Computing Environment (UCE). This chapter discusses how this thesis has built on related work in the area of information systems, agents, the World Wide Web, and other Pervasive and Ubiquitous Computing projects. This work was the necessary foundation, from which this thesis developed theoretical and practical techniques needed for UCE.

The composition algorithm developed in this thesis is the critical piece needed to enable the dynamic construction of a local ontology. The semantic query algorithm defined in this thesis is the foundation for more intelligent services and interfaces. Together, these an important step to constructing better Ubiquitous Computing Environment.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

References

1. Abowd, Gregory D. and Elizabeth D Mynatt, *Charting Past, Present, and Future Research in Ubiquitous Computing.* ACM Transactions on Computer-Human Interaction, *7 (1):29-58*, 2000.
2. Ankolekar, Anupriya, Mark Burnstein, Jerry R. Hobbs, Ora Lassila, David Martin, Drew McDermott, Sheila A. McIlraith, Srini Narayanan, Massimo Paolucci, Terry Payne, and Katia Sycara. *DAML-S: Web Service Description for the Semantic Web.* In *First International Semantic Web Conference (ISWC)*, 2002, 348-363.
3. Antoniou, Grigoris. *Nonmonotonic Rule Systems on Top of Ontology Layers.* In *International Semantic Web Conference*, 2002, 394-398.
4. Baader, Franze, Diego Calvenese, Deborah McGuiness, Daniele Nardi, and Peter Patel-Schneider, eds. *The Description Logic Handbook: theory, implementation, and applications.* Cambridge University Press: Cambridge,2003.
5. Bagüés, Mirin I., Jesus Bermúdez, Arantza Illarramendi, A. Tablado, and Alfredo Goñi. *Using Ontologies in the Development of an Innovating System for Elderly People Tele-Assistance.* In *International Conference on Ontologies, Databases, and Applications of Semantics (OODBASE 2003)*, 2003, 889-905.
6. Bansal, Sharad and José M. Vidal. *Matchmaking of Web Services Based on the DAML-S Service Model.* In *Autonomous Agents and Multiagent Systems*, 2003, 926-927.
7. Bechhofer, Sean, Carole Goble, and Ian Horrocks. *DAML+OIL is not enough.* In *Second International Workshop on the Semantic Web*, 2001, 151-160.
8. Bechhofer, Sean, Ian Horrocks, Carole Gable, and Robert Stevens. *OilEd: a Reason-able Ontology Editor for the Semantic Web.* In *KI2001: Advances in Artificial Intelligence*, 2001.
9. Bechhofer, Sean, Ian Horrocks, Peter F. Patel-Schneider, and Sergio Tessaris. *A proposal for a description logic interface.* In *International Workshop on Description Logics (DL'99)*, 1999, 33-36.
10. Bechhofer, Sean, Ian Horrocks, and Sergio Tessaris, *CORBA interface for a DL Classifier.* 1999.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

11. Beck, Howard W., Tarek Anwar, and Shamkant B. Navathe, *A Conceptual Clustering Algorithm for Database Schema Design.* IEEE Transactions on Knowledge and Data Engineering, *6 (3):396-410*, 1994.

12. Bellwood, Tom, Luc Clement, and Claus von Riegen, *UDDI Version 3.0.1.* OASIS UDDI Spec Technical Committee Specification, 2003. http://uddi.org/pubs/uddi_v3.htm

13. Berners-Lee, T., R. Fielding, and L. Masinter, *Uniform Resource Identifiers (URI): Generic Syntax.* IETF RFC 2396, 1998. http://www.ietf.org/rfc2398.txt

14. Berners-Lee, Tim, James Hendler, and Ora Lassila, *The Semantic Web.* Scientific American, *284 (5):35-43*, 2001.

15. BFO/MedO. *Basic Formal Ontology and Medical Ontology Draft 0.0006.* 2003, http://ontology.buffalo.edu/bfo/BFO.html.

16. Borgida, Alex and Ronald J. Brachman, *Loading Data into Description Reasoners.* ACM SIGMOD Record, *22 (2):217-226*, 1993.

17. Borgida, Alex and Luciano Serafini, *Distributed Description Logics: Assimilating Information From Peer Services.* Journal on Data Semantics, *1 (1):153-184*, 2003.

18. Borgida, Alexander, *Description Logics in Data Management.* IEEE Transactions on Knowledge and Data Engineering, *7 (5):671-682*, 1995.

19. Bryson, Joanna J., David L. Martin, Sheila A. McIlraith, and Lynn Andreas Stein, *Toward Behavioral Intelligence in the Semantic Web.* IEEE Computer, *35 (11):48-54*, 2002.

20. Calvanese, Diego, Giuseppe De Giacomo, and Maurizio Lenzerini. *Description Logics: Foundations for Class-based Knowledge Representation.* In *IEEE Symposium on Logic in Computer Science*, 2002, 359-370.

21. Calvanese, Diego, Mauritzio Lenzerini, and Daniele Nardi, *Unifying Class-based Representation Formalisms.* Journal of Artificial Intelligence Research, *11:199-240*, 1999.

22. Ceri, Stefano, Georg Gottlob, and Letizia Tanca, *Logic Programming and Databases*, Berlin, Springer-Verloag, 1990.

23. Cerqueira, Renato, Christopher K. Hess, Manuel Roman, and Roy H. Campbell. *Gaia: A Development Infrastructure for Active Spaces.* In *Workshop on Application Models and Programming Tools for Ubiquitous Computing*, 2001.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

24. Cerqueira, Renato, Cristina Ururahy, Christopher K. Hess, Dulcinea Carvalho, Manuel Roman, Noemi Rodriguez, and Roy H. Campbell. *Support for Mobility in Active Spaces*. In *Workshop on Middlewar for Mobile Computing*, 2001.
25. Chakraborty, Dipanjan, Filip Perich, Sasikanth Avancha, and Anupam Joshi. *DReggie: Semantic Service Discovery for M-Commerce Applications*. In *Symposium on Reliable Distributed Systems*, 2001.
26. Chaudhri, Vinay K., Adam Farquhar, Richard Fikes, Peter D. Karp, and James P. Rice, *Open Knowledge Base Connectivity 2.0.3--Proposed.* Stanford Research Institute, 1998.
27. Chen, Guanlin and David Kotz. *Context-Sensitive Resource Discovery*. In *First International Conference on Pervasive Computing and Communications*, 2003, 243-252.
28. Chen, Harry, *Personal Communication*. 2003.
29. Chen, Harry, Tim Finin, and Anupam Joshi. *Semantic Web in a Pervasive Context-Aware Architecture*. In *Artificial Intelligence in Mobile Systems workshop at Ubicomp 2003*, 2003.
30. Chen, Harry, Tim Finin, and Anupam Joshi. *Using OWL in a Pervasive Computing Broker*. In *Workshop on Ontologies in Agent Systems (OAS'03) at Autonomous Agents and Multi-Agent Systems*, 2003, 1-24.
31. Chen, Harry, Tim Finin, Anupam Joshi, Lalana Kagal, Filip Perich, and Dipanjan Chakraborty, *Intelligent Agents Meet the Semantic Web in Smart Spaces.* IEEE Internet Computing, *8 (6):69-79*, November/December 2004.
32. Chen, Harry, Filip Perich, Tim Finin, and Anupam Joshi. *SOUPA: Standard Ontology for Ubiquitous and Pervasive Applications*. In *MOBIQUITOUS 2004 - 1st Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services*, 2004, 258-267.
33. Chen, Hsinchun, Tobun D. Ng, Joanna Martinez, and Bruce R. Schatz, *A Concept Space Approach to Addressing the Vocabulary Problem in Scientific Information Retrieval: An Experiment on the Worm Community System.* Journal American Society Information Science (JASIS), *48 (1):17-31*, 1997.
34. Chen, L., N. R. Shadbolt, F. Tao, S. J. Cox, A. J. Keane, C. Goble, A. Roberts, and P. Smart. *Engineering Knowledge for Engineering Grid Applications*. In *Euroweb 2002 -- The Web and the GRID: from e-science to e-business*, 2002.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

35. Chen, Siming, Nigel R. Shadbolt, Carole Goble, Fen Tao, Simon J. Cox, Colin Puleston, and P. R. Smart. *Towards a Knowledge-Based Approach to Semantic Service Composition*. In *Second International Semantic Web Conference*, 2003, 319-334.

36. CHI 2000. *Workshop 11: The What, Who, Where, When, Why and How of Context-Awareness*. 2000, http://www1.acm.org/sigs/sigchi/chi2000/advance-program/.

37. Christensson, Bengt and Olof Larsson. *Universal Plug and Play Connects Smart Devices*. In *WinHEC 99*, 1999.

38. Connelly, Kay and Ashraf Khalil. *Towards Automatic Device Configuration in Smart Environments*. In *Ubisys workshop at Ubicomp 2003*, 2003.

39. Cotter, Sean and Mike Potel, *Inside Taligent Technology*, Reading, MA, Addison-Wesley, 1995.

40. Czerwinski, Steven E., Ben Y. Zhao, Todd D. Hodes, Anthony D. Joseph, and Randy H. Katz. *An Architecture for a Secure Service Discovery Service*. In *Mobicom'99*, 1999, 24-35.

41. daml.org. *A DAML Ontology of Time*. 2002, http://www.cs.rochester.edu/~ferguson/daml/daml-time-29jul02.txt.

42. daml.org. *The DARPA Agent Markup Language Homepage*. 2003, http://www.daml.org.

43. De Roure, David, Nicholas R. Jennings, and Nigel R. Shadbolt, *The Semantic Grid: Past, Present, and Future*. Proceedings of the IEEE, *93 (3):669-681*, 2005.

44. Decker, Stefan, Michael Erdmann, Dieter Fensel, and R. Studer, *Ontobroker: Ontology based access to distributed and semi-structured information*, in *Semantic Issues in Multimedia Systems*, R. Meersman, Z. Tari, and S. Stevens, Editors. Kluwer, Boston, 1999.

45. Decker, Stefan, Dieter Fensel, Frank van Harmelen, Ian Horrocks, Sergey Melnik, Michel Klein, and Jeen Broekstra. *Knowledge Representation on the Web*. In *International Workshop on Description Logics*, 2000.

46. Description Logic Implementation Group. *DIG*. 2003, http://dl.kr.org/dig.

47. Dey, Anind K., Daniel Salber, Masayasu Futakawa, and Gregory D. Abowd, *An Architecture to Support Context-Aware Applications*. GVU Technical Report GIT-GVU-99-23, 1999. ftp://ftp.gvu.gatech.edu/pub/gvu/tr/99-23.pdf

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

48. Dickinson, Robert D., *Object oriented system for representing physical locations.* 1997, Object Technology Licensing Corp.: United States.

49. Dickinson, Robert David, *Method and apparatus for displaying business cards.* 1998: United States.

50. Doan, AnHai, Jayant Madhavan, Robin Dhamankar, Pedro Domingos, and Alon Halevy, *Learning to Match Ontologies on the Semantic Web.* The VLDB Journal, *12 (4):303-319*, 2003.

51. Doan, AnHai, Jayant Madhavan, Pedro Domingos, and Alon Halevy. *Learning to Map between Ontologies on the Semantic Web.* In *WWW2002*, 2002.

52. Dou, Dejing, Drew McDermott, and Peishen Qi. *Ontology Translation on the Semantic Web.* In *Internationall Conference on Ontologies, Databases and Applications of Semantics (ODBSAE2003)*, 2003.

53. Edwards, W. Keith, *Core JINI*, Upper Saddle River, NJ, Prentice Hall, 1999.

54. EuroWordNet. *EuroWordNet.* http://www.illcc.uva.nl/EuroWordNet/.

55. Eustice, K. F., T. J. Lehman, A. Morales, M. C. Munson, S. Edlund, and M. Guillen, *A universal information appliance.* IBM Systems Journal, *38 (4)* 1999.

56. Fensel, Dieter, *Ontologies: a silver bullet for knowledge management and electronic commerce*, Berlin, Springer, 2001.

57. Fensel, Dieter, *Ontology-Based Knowledge Management.* IEEE Computer, *35 (11):56-59*, 2002.

58. Fensel, Dieter, Cristoph Bussler, and Alexander Maedche. *Semantic Web Enabled Web Services.* In *International Semantic Web Conferences*, 2002, 1-2.

59. Fensel, Dieter, Ian Horrocks, Frank van Harmelen, Deborah L. McGuinness, and Peter F. Patel-Schneider, *OIL: An Ontology Infrastructure for the Semantic Web.* IEEE Intelligent Systems, *16 (2):38-45*, 2001.

60. Fikes, Richard, Gleb Frank, and Jessica Jenkins, *JTP: A System Architecture and Component Library for Hybrid Reasoning.* Stanford University Knowledge Systems Lab KSL-03-01, 2003. http://ksl.stanford.edu/KSL_Abstracts/KSL-03-01.html

61. Fikes, Richard, Pat Hayes, and Ian Horrocks. *DAML Query Language (DQL): Abstract Specification (April 2003).* 2003, http://www.daml.org/2003/dql/dql.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

62. Fikes, Richard, Pat Hayes, and Ian Horrocks. *DQL-A Query Language for the Semantic Web.* In *WWW 2003*, 2003.

63. Fikes, Richard and Deborah I. McGuinness. *An Axiomatic Semantics for RDF, RDF-S, and DAML+OIL.* 2001, http://www.daml.org/2001/03/axiomatic-semantics-071601.html.

64. Foundation for Intelligent Physical Agents. *Foundation for Intelligent Physical Agents.* http://www.fipa.org.

65. Fox, Armando, Brad Johanson, Pat Hanrahan, and Terry Winograd, *Integrating Information Appliances into an Interactive Workspace.* IEEE Computer Graphics and Applications, *20 (3):54-65*, 2000.

66. Franconi, Enrico. *Description Logics and Logics.* http://www.cs.man.ac.uk/~franconi/dl/course.

67. Franconi, Enrico. *Propositional Description Logics.* http://www.cs.man.ac.uk/~franconi/dl/course/propositional-dl.ps.gz.

68. Furnas, George W., Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, Richard A. Harshman, Lynn A. Streeter, and Karen E. Lochbaum. *Information Retrieval Using A Singular Value Decomposition Model of Latent Semantic Structure.* In *Eleventh International ACM/SIGIR Conference on Research and Development in Information Retrieval*, 1988, 465-480.

69. Gallaire, Herve, Jack Minker, and Jean-Marie Nicolas, *Logic and Databases: A Deductive Approach.* ACM Computing Surveys, *16 (2):153-185*, 1984.

70. Garlan, David, Daniel P. Siewiorek, Asim Smailagic, and Peter Steenkiste, *Project Aura: Towards Distraction-Free Pervasive Computing.* IEEE Pervasive Computing, *1 (2):22-31*, 2002.

71. Genesereth, Michael R. *Knowledge Interchange Format: draft proposed American National Standard (dpANS).* 1999, http://logic.stanford.edu/kif/dpans.html.

72. Glaser, John P. and Leslie D. Hsu, *The Strategic Application of Information Technology in Healthcare Organizations*, New York, McGraw-Hill, 1999.

73. Globus. *The Globus Resource Specification Language.* http://www-fp.globus.org/gram/rsl_spec1.htm.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

74. Goasdoué, François and Marie-Christine Rousset, *Querying Distributed Data through Distributed Ontologies: A Simple but Scalable Approach.* IEEE Intelligent Systems, *18 (5):60-65*, 2003.

75. Goble, Carole and David De Roure, *The Grid: An Application of the Semantic Web.* ACM SIGMOD Record, *31 (4):65-70*, 2002.

76. GoF Design Patterns. *Proxy*. 2003, http://www.tml.hut.fi/Tik-76.278/gof/html/Proxy.html.

77. Golbreich, Christine, Olivier Dameron, Bernard Gibaud, and Anita Burgun. *Web Ontology Language Requirements w. r. t. Expressiveness of Taxonomy and Axioms in Medicine*. In *Second International Semantic Web Conference*, 2003, 180-194.

78. Goñi, Alfredo, Arantza Illaramendi, and José Miguel Blanco, *Caching in Multidatabase Systems based on DL.* 2000.

79. Goñi, Alfredo, Arantza Illaramendi, Eduardo Mena, and José Miguel Blanco, *An Ontology Connected to Several Data Repositories: Query Processing Steps.* Journal of Computing and Information, *3 (1):-*, 1998.

80. Gonzalez-Castillo, Javier, David Trastour, and Claudio Bartolini, *Description Logics for Matchmaking Services*. Bristol, HP Laboratories Bristol HPL-2001-265, Bristol, 2002. http://www.hpl.hp.com/techreports/2001/HPL-2001-265.html

81. Greiner, Russell, Christian Darken, and N. Iwan Santoso, *Efficient Reasoning.* ACM Computing Surveys, *33 (1):1-30*, 2001.

82. Grenen, Pierre and Barry Smith, *SNAP and SPAN: Towards Dynamic Spatial Ontology.* Spatial Cognition and Computation, *(forthcoming)* 2004.

83. Gribble, Steven D., Matt Welsh, Rob von Behren, Eric A. Brewer, David Culler, N. Borisov, S. Czerinski, R. Bummadi, J. Hill, A. Joseph, R. H. Katz, Z. M. Mao, S. Ross, and B. Ziao, *The Ninja Architecture for Robust Internet-Scale Systems and Services.* Computer Networks, 2000.

84. Grigson, Geoffrey and Charles Harvard Gibbs-Smith, *Things*. People, Places and Things, ed. G. Grigson and C.H. Gibbs-Smith. Vol. 3, London, The Grosvenor Press, 1954.

85. Grosof, Benjamin N and Ian Horrocks, *Description Logic Programs: Combining Logic Programs with Description Logic*. 2002.

86. Grosof, Benjamin N., Yannis Labrou, and Hoi Y. Chan. *A Declarative Approach to Business Rules in Contracts: Courteous Logic Programs in XML.* In *E-COMMERCE*, 1999, 68-77.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

87. Grosof, Benjamin N. and Terrence C. Poon. *Representing Agent Contracts with Exceptions using XML Rules, Ontologies, and Process Descriptions*. In *Workshop on Rule Markup Languages for Business Rules, at International Semantic Web Conference*, 2002.

88. Gruber, Thomas R., *A translation approach to portable ontologies.* Knowledge Acquisition, *5 (2):199-200*, 1993.

89. Gu, Xiaohui, Klara Nahrstedt, Wanghong Yuan, Duangdao Wichadakul, and Dongyan Xu, *An XML-based Quality of Service Enabling Language for the Web.* Journal of Visual Language and Computing, *13 (1):61-75*, 2002.

90. Guarino, Nicola. *Formal Ontology and Information Systems*. In *Formal Ontology and Information Systems*, 1998, 3-15.

91. Guttman, E., C. Perkins, J. Veizades, and M. Day, *Service Location Protocol, Version 2.* IETF RFC 2608, 1999. http://www.rfc-editor.org/rfc/rfc2608.txt

92. Guttman, Erik, *Service Location Protocol: Automatic Discovery of IP Network Services.* IEEE Internet Computing, *3 (4):71-80*, July-August 1999.

93. Haarslev, Volker and Ralf Möeller. In *Seventeenth International Joint Conference on Artificial Intelligence*, 2001.

94. Haarslev, Volker and Ralf Möller. *High Performance Reasoning with Very Large Knowledge Bases: A Practical Case Study*. In *Seventeenth International Conference on Artificial Intelligence*, 2001.

95. Haarslev, Volker and Ralf Möller. *RACER System Description.* In *Automated Reasoning: First International Joint Conference*, 2001, 701.

96. Haarslev, Volker and Ralf Möller. *RACER User's Guide and Reference Manual.* 2003, http://www.cs.concordia.ca/~haarslev/racer.

97. Halevy, Alon Y., Zachary G. Ives, Peter Mork, Dan Suciu, and Igor Tatarinov, *The Piazza Peer Data Management System.* IEEE Transactions on Knowledge and Data Engineering, *16 (7):787-798*, July 2004.

98. Halevy, Alon Y., Zachary G. Ives, Dan Sucio, and Igor Tatarinov. *Schema Mediation in Peer Data Management Systems*. In *ICDE 2003*, 2003.

99. Harle, Robert K. and Andy Hopper. *Building World Models by Ray-Tracing*. In *UbiComp 2003: Ubiquitous Computing*, 2003, 1-17.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

100. Harmelen, Frank van, Peter F. Patel-Schneider, and Ian Horrocks. *A Model-Theoretic Semantics for DAML+OIL*. 2001, http://www.daml.org/2001/03/model-theoretic-semantics.html.

101. Harmelon, Frank, Peter F. Pael-Schneider, and Ian Horrocks. *Annotated DAML+OIL (March 2001) Ontology Markup*. 2001, http://www.daml.org/2001/03/daml+oil-walkthru.html.

102. Harmelon, Frank van, Peter F. Patel-Schneider, and Ian Horrocks. *Reference description of the DAML+OIL (March 2001) ontology markup language*. 2001, http://www.daml.org/2001/03/reference.html.

103. Hayes, Patrick, *RDF Semantics*. W3C W3C Recomendataion rdf-mt, 2004. http://www.w3.org/TR/rdf-mt

104. Henning, Michi and Steve Vinoski, *Advanced CORBA Programming with C++*, Addison Wesley, 1999.

105. Hess, Christopher K., Manual Roman, and Roy H. Campbell. *Building Applications for Ubiquitous Computing Environments*. In *International Conference on Pervasive Computing*, 2002, pp. 16-29.

106. Hoffman, Kevin, Jeff Gabriel, Denise Gosnell, Jeff Hasan, Cristian Holm, Ed Musters, Jan Narkiewickz, John Schenken, Thiru Thangarathinam, Scott Wylie, and Jonothan Ortiz, *Professional.NET Framework*, Birmingham, WROX Press Ltd., 2001.

107. Horrocks, Ian. *CORBA-FaCT*. http://www.cs.man.ac.uk/~horrocks/FaCT/CORBA-FaCT.html.

108. Horrocks, Ian. *The FaCT system*. In *Automated Reasoning with Analytic Tableaux and Related Methods*, 1998, 307-312.

109. Horrocks, Ian, *Optimising Tableaux Decision Procedures for Description Logics*. 1999, Manchester.

110. Horrocks, Ian. *A Denotational Semantics for Standard OIL and Instance OIL*. 2000, http://www.ontoknowledge.org/oil/downl/semantics.pdf.

111. Horrocks, Ian, *Reasoning with Expressive Description Logics: Theory and Practice*. 2001, University of Leipzig.

112. Horrocks, Ian and Peter F. Patel-Schneider, *Optimizing Description Logic Subsumption*. Journal of Logic and Computation, *9 (3):267-293*, 1999.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

113. Horrocks, Ian and Peter F. Patel-Schneider. *Reducing OWL Entailment to Description Logic*. In *Second International Semantic Web Conference*, 2003, 17-29.

114. Horrocks, Ian and Peter F. Patel-Schneider. *A proposal for an owl rules language*. In *Thirteenth International World Wide Web Conference (WWW 2004)*, 2004, 723-731.

115. Horrocks, Ian, Peter F. Patel-Schneider, and Frank van Harmelen, *From SHIQ and RDF to OWL: The making of a web ontology language.* Journal of Web Semantics, *1 (1):7-26*, 2003.

116. Horrocks, Ian and Ulrike Sattler. *Ontology Reasoning with SHOQ(D) Description Logic*. In *International Joint Conference on Artificial Intelligence*, 2001, 199-204.

117. Horrocks, Ian, Ulrike Sattler, and Stephan Tobias. *Practical Reasoning for Expressive Description Logics*. In *International Conference on Logic for Programming and Automated Reasoning (LPAR'99)*, 1999, 161-180.

118. Horrocks, Ian and Sergio Tessaris. *Querying the Semantic Web: A Formal Approach*. In *International Semantic Web Conference*, 2002, 177-191.

119. Huang, Andrew C., Benjamin C. Ling, John Barton, and Armando Fox. *Making Computers Disappear: Appliance Data Services*. In *Seventh ACM/IEEE International Conference on Mobile Computing (MobiCom 2001)*, 2001.

120. Huston. *Proxy*. 2003, http://rampages.onramp.net/~huston/dp/proxy.html.

121. IEEE P1600.1 Standard Upper Ontology Working Group. *Standard Upper Ontology (SUO) Working Group*. 2003, http://suo.ieee.org.

122. International Standard Serial Number. *ISSN Home Page*. http://www.issn.org.

123. Johanson, Brad and Armando Fox. *The EventHeap: A Coordination Infrastructure for Interactive Workspaces*. 2001, http://graphics.stanford.edu/~bjohanso/papers/ubicomp2001/eheap_ubicomp.pdf.

124. Kagal, Lalana, Tim Finin, and Anupam Joshi. *A Policy Language for a Pervasive Computing Environment*. In *Fourth IEEE International Workshop on Policies for Distributed Systems and Networks*, 2003.

125. Kappel, Gertl, Stefan Rausch-Scott, and Werner Retschitzegger, *A framework for workflow management systems based on objects, rules and roles.* ACM Computing Surveys, *32 (1es)* 2000.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

126. Kesselman, Carl. *The Grid, Grid Services and the Semantic Web.* In *International Semantic Web Conference*, 2002, 3-4.

127. Kidd, Alison L., ed. *Knowledge Acquisition for Expert Systems.* Plenum: New York,1987.

128. Kifer, Michael, Georg Lausen, and James Wu, *Logical Foundations of Object-Oriented and Frame-Based Languages.* Journal of the Association for Computing Machinery, *42 (4):741-843*, 1995.

129. Kim, Dan Jong, Manish Agrawal, Bhara Jayaraman, and H. Raghav Rao, *A Comparison of B2B E-Service Solutions.* Communications of the ACM, *46 (12e):317-324*, 2003.

130. Kindberg, Tim, John Barton, Jeff Morgan, Gene Becker, Ilja Bedner, Debbie Caswell, Phillipe Debaty, Gita Gopal, Marcos Frid, Venky Krishnan, Howard Morris, Celene Pering, John Schettino, and Bill Serra. *People, Places, Things: Web Presence for the Real World.* In *WWW'2000*, 2000.

131. Koile, Kimberle, Konrad Tollmar, David Demirdjian, Howard Shrobe, and Trevor Darrell. *Activity Zones for Context-Aware Computing.* In *UbiComp 2003: Ubiquitous Computing*, 2003, 90-106.

132. Kon, Fabio, *Automatic Configuration of Component-Based Systems*, in *Computer Science.* 2000, University of Illinois, Urbana-Champaign: Urbana.

133. Kon, Fabio, Tomonori Yamane, Christopher Hess, Roy Campbell, and M. Dennis Mickunas. *Dynamic Resource Management and Automatic Configuration of Distributed Component Systems.* In *The 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'2001)*, 2001.

134. Korfhage, Robert R., *Information Storage and Retrieval*, New York, John Wiley and Sons, Inc., 1997.

135. Lampson, Butler, Martin Abadi, Michael Burrows, and Edward Wobber, *Authentication in Distributed Systems: Theory and Practice.* ACM Transactions of Computer Systems, *10 (4):265-310*, 1992.

136. Lancaster, F. W., *Vocabulary Control for Information Retrieval*, Arlington, VA, Information Retrieval Press, 1986.

137. Leonhardt, Ulf, *Supporting Location Awareness in Open Distributed Systems*, in *Computing.* 1998, Imperial College London: London.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

138. Levas, Anthony, Claudio Pinhanez, Gopal Pingali, Rick Kjeldsen, Mark Podlaseck, and Noi Subaviriya. *An Architecture and Framework for Steerable Interface Systems*. In *UbiComp 2003: Ubiquitous Computing*, 2003, 333-348.

139. Li, Lei and Ian Horrocks. *A software framework for matchmaking based on semantic web technology*. In *Twelfth International World Wide Web Conference (WWW 2003)*, 2003, 331-339.

140. Liu, Mengchi, *Deductive Database Languages: Problems and Solutions.* ACM Computing Surveys, *31 (1):27-62*, 1999.

141. Lyytinen, Kalle and Youngjin Yoo, *Issues and Challenges in Ubiquitous Computing.* CACM, *45 (12):62-65*, 2002.

142. Martin, David, Mark Burstein, Jerry Hobbs, Ona Lassila, Drew McDermott, Sheila McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, Evren Sirin, Naveen Srinivasen, and Katia Sycara, *OWL-S: Semantic Markup for Web Services.* W3C Member Submission, 2004. http://www.w3.org/Submission/2004/SUBM-OWL-S-20041122/

143. Masuoka, Ryusuke, Yannis Labrou, Bijan Parsia, and Evren Sirin, *Ontology-Enabled Pervasive Computing Applications.* IEEE Intelligent Systems, *18 (5):68-72*, 2003.

144. McGrath, Robert E., *Discovery and Its Discontents: Discovery Protocols for Ubiquitous Computing*. Urbana, Department of Computer Science University of Illinois Urbana-Champaign UIUCDCS-R-99-2132, Urbana, 2000. http://www.cs.uiuc.edu/research/techreports.php?report=UIUCDCS-R-2000-2154

145. McGrath, Robert E., *Challenge Problems for Pervasive Computing (presentation to Gaia group)*. 2003.

146. McGrath, Robert E., *A Model for Spatial Properties and Services in a Ubiquitous Computing Environment (to appear)*. 2003.

147. McGrath, Robert E., Joe Futrelle, Ray Plante, and Damien Guillaume. *Digital Library Technology for Locating and Accessing Scientific Data*. In *ACM Digital Libraries '99*, 1999, 188-194.

148. McGrath, Robert E. and M. Dennis Mickunas. *Dynamic Personal Roles For Ubiquitous Computing (Poster)*. In *OOPSLA*, 2003.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

149. McGrath, Robert E. and M. Dennis Mickunas, *Dynamic Roles: Organizing Software Representatives for People*. Urbana, Department of Computer Science UIUCDCS-R-2003-2355 UILU-ENG-2003-1742, Urbana, 2003.

150. McGrath, Robert E., Anand Ranganathan, Roy H. Campbell, and M. Dennis Mickunas. *Incorporating "Semantic Discovery" into Ubiquitous Computing Environments*. In *Ubisys 2003*, 2003.

151. McGrath, Robert E., Anand Ranganathan, Roy H. Campbell, and M. Dennis Mickunas, *Use of Ontologies in Pervasive Computing Environments*. Urbana, Department of Computer Science UIUCDCS-R-2003-2332 UILU-ENG-2003-1719, Urbana, 2003. http://www.cs.uiuc.edu/research/techreports.php?report=UIUCDCS-R-2003-2332

152. McGrath, Robert E., Anand Ranganathan, M. Dennis Mickunas, and Roy H. Campbell. *Investigations of Semantic Interoperability in Ubiquitous Computing Environments*. In *International Conference on Parallel and Distributed Computing Systems*, 2003.

153. McGuinness, Deborah L., *Explaining Reasoning in Description Logics*, in *Computer Science*. 1996, Rutgers: New Brunswick.

154. McGuinness, Deborah L. *Ontology-enhanced Search for Primary Care Medical Literature*. In *Medical Concept Representation and Natural Language Processing*, 1999.

155. McGuinness, Deborah L., *Ontologies and Online Commerce.* IEEE Intelligent Systems, *16 (1):8-14*, 2001.

156. McGuinness, Deborah L., Richard Fikes, James Rice, and Steve Wilder. *An Environment for Merging and Testing Large Ontologies*. In *International Conference on Principles of Knowledge Representation and Reasoning*, 2000.

157. McGuinness, Deborah L. and Paulo Pinheiro da Silva. *Infrastructure for Web Explanations*. In *Second International Semantic Web Conference*, 2003, 113-129.

158. McGuinness, Deborah L. and Jon R. Wright, *An Industrial-Strength Description Logic-Based Configurator Platform.* IEEE Intelligent Systems, *13 (4):69-77*, 1998.

159. McIlraith, Sheila A., Tran Cao Son, and Honlei Zeng, *Semantic Web Services.* IEEE Intelligent Systems, *16 (2):46-53*, 2001.

160. Medin, Douglas L. and Scot Atran, eds. *Folkbiology*. MIT Press: Cambridge,1990.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

161. Mena, Eduardo, Arantza Illarramendi, V. Kashap, and A. Sheth, *OBSERVER: An Approach for Query Processing in Global Information Systems Based on Interoperation Across Pre-existing Ontologies.* International Journal of Distributed and Parallel Databases, *8 (2):223-272*, 2000.

162. Microsoft Corporation. *Universal Plug and Play: Background.* http://www.upnp.org/resources/UpnPbkgnd.htm.

163. Microsoft Corporation, *Universal Plug and Play Device Architecture Reference Specification.* Microsoft Corporation, 1999. http://www.microsoft.com/hwdev/UPnP

164. Minsky, Marvin, *A Framework for Representing Knowledge*, in *The Psychology of Computer Vision*, P. Winston, Editor. McGraw Hill, New York, 1975.

165. Mockapetris, P., *Domain Names-Implementation and Specification.* IETF RFC 1035, 1987. http://www.rfc-editor.org/rfc/rfc1035.txt

166. National Library of Medicine. *Unified Medical Language System (UMLS).* http://www.nlm.nih.gov/research/umls/.

167. Niles, Ian and Adam Pease. *Origins of the IEEE Standard Upper Ontology.* In *IJACAI Workshop on the IEEE Standard Upper Ontology*, 2001.

168. Noia, Tommaso Di, Eugenio Di Sciascio, Francesco M. Donini, and Mariana Mongiello. *A System for Principled Matchmaking in an Electronic Marketplace.* In *WWW2003*, 2003, 321-330.

169. Norman, Donald A., *The Invisible Computer*, Cambridge, MA, MIT Press, 1998.

170. Noy, Natalya F., Michael Sintek, Stefan Decker, Monica Crubezy, Ray W. Fergerson, and Mark A. Musen, *Creating Semantic Web Contents with Protégé-2000.* IEEE Intelligent Systems, *16 (2):60-71*, 2001.

171. Noy, Natalya Fridman, Ray W. Fergerson, and Mark A. Musen. *The Knowledge Model of Protégé-2000: Combining Interoperability and Flexibility.* In *Twelfth International Conference on Knowledge Engineering and Knowledge Management*, 2000.

172. OASIS. *Current OASIS Technical Committees.* http://www.oasis-open.org/committees/committees.html.

173. Oberle, Daniel, Rapael Volz, Steffen Staab, and Boris Motik, *An Extensible Open Software Environment*, in *International Handbooks on Information Systems.* Springer, 2003, 311-333.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

174. Object Management Group. *TC Plenaries and Subgroup Directory.* http://www.omg.org/technology/documents/domain_spec_catalog.htm.

175. Object Management Group, *CORBAservices: Common Object Services Specification.* Object Management Group, 1999. ftp://ftp.omg.org/pub/.docs/formal/98-07-05.pdf

176. Object Management Group, *Trading Service Specification.* 2000.

177. Object Management Group, *Bibliographic Query Service Specification.* 2002.

178. OilEd. *OilEd.* 2002, http://oiled.man.ac.uk/.

179. Ontomap.org. *Ontomap.org.* http://www.ontomap.org.

180. OntoMerge. *OntoMerge: Ontology Translation by Merging Ontologies.* http://cs-www.cs.yale.edu/homes/dvm/daml/ontology-translation.html.

181. ontopriseGMB. *Ontoprise: Semantics for the Web.* 2003, www.ontoprise.com.

182. OpenCyc.org. *OpenCyc.org: Formalized Common Knowledge.* http://www.opencyc.org/.

183. Pan, Jeff Z. and Ian Horrocks. *Reasoning in the SHOQ(D) Description Logic.* In *Workshop on Description Logics (DL-2002)*, 2002, 53-62.

184. Paolucci, Massimo, Takahiro Kawamura, Terry R. Payne, and Katia Sycara. *Semantic Matching of Web Service Capabilities.* In *First International Semantic Web Conference*, 2002, 333-347.

185. Partridge, Chris, *Business Objects: Engineering for Re-use*, Oxford, Butterworth Heinemann, 1996.

186. Partridge, Chris, *The Role of Ontology in Integrating Semantically Heterogeneous Databases.* Padova, LASSEB-CNR Technical Report 05/02, Padova, 2002.

187. Pascoe, Bob, *Salutation Architectures and the newly defined service discovery protocols from Microsoft and Sun.* Salutation Consortium White Paper, 1999. http://www.salutation.org/whitepaper/JINI-UPnP

188. Pascoe, Jason, Nick Ryan, and David Morse. *Issues in Developing Context-Aware Computing.* In *HandHeld and Ubiquitous Computing*, 1999, 208-221.

189. Peer, Joachim. *Bringing Together Semantic Web and Web Services.* In *First International Semantic Web Conference (ISWC)*, 2002, 279-291.

190. Ponnekanti, Shankar R., Brian Lee, Armando Fox, Pat Hanrahan, and Terry Winograd. *ICrafter: A Service Framework for Ubiquitous Computing Environments.* In *UIBCOMP 2001*, 2001.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

191. Quillian, M. Ross, *Semantic Networks*, in *Semantic Information Processing*, M. Minsky, Editor. MIT Press, Cambridge, 1968.

192. Quine, W. V. O., *On What There Is*, in *From a Logical Point of View*. Harper & Row, New York, 1953.

193. Rahm, Erhard and Phillip A. Bernstein, *A survey of approaches to automatic schema matching*. The VLDB Journal, *10:334-350*, 2001.

194. Ramakrishnan, Raghu, ed. *Applications of Logic Databases*. Kluwer: Boston,1995.

195. Ramakrishnan, Raghu and Jeffrey D. Ullman, *A Survey of Deductive Database Systems*. Journal of Logic Programming, *23 (2):127-149*, 1994.

196. Raman, Rajeesh, Miron Livny, and Marv Solomon, *Matchmaking: An extensible framework for distributed resource management*. Cluster Computing, *2 (2):126-138*, 1999.

197. Ranganathan, Anand. *Space Repository and Active Space Web Browser*. 2001, http://devius.cs.uiuc.edu/2k/Gaia/doc/dev/SpaceRepository.pdf.

198. Ranganathan, Anand and Roy Campbell, *An Infrastructure for Context-Awareness Based on First Order Logic*. Personal and Ubiquitous Computing, *7 (6):353-364*, 2003.

199. Ranganathan, Anand and Roy H. Campbell. *A Middleware for Context-Aware Agents in Ubiquitous Computing*. In *ACM/IFIP/Usenix Middleware*, 2003.

200. Ranganathan, Anand, Shiva Chetan, and Roy Campbell. *Mobile Polymorphic Applications in Ubiquitous Computing Environments*. In *MOBIQUITOUS 2004 - 1st Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services, Proceedings of MOBIQUITOUS 2004*, 2004, 402-411.

201. Ranganathan, Anand, Robert E. McGrath, Roy Campbell, and M. Dennis Mickunas. *Ontologies in a Pervasive Computing Environment*. In *Workshop on Ontologies in Distributed Systems at IJCAI*, 2003-dup.

202. Ranganathan, Anand, Robert McGrath, Roy Campbell, and Dennis Mickunas. *Ontologies in a Pervasive Computing Environment*. In *Workshop on Ontologies in Distributed Systems at International Joint Conference on Artificial Intelligence*, 2003.

203. Ranganathan, Anand, Robert E. McGrath, Roy H. Campbell, and M. Dennis Mickunas, *Use of Ontologies in a Pervasive Computing Environment*. Knowledge Engineering Review, *18 (3):209-220*, 2004.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

204. Reynolds, Dave, *Semantic Web Chalk Talk: Amateur Intro to Decription Logics*. 2001, HP Laboratories: Bristol.

205. Roman, Manuel, Christopher K. Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt, *Gaia: A Middleware Infrastructure to Enable Active Spaces*. IEEE Pervasive Computing, *1 (4):74-83*, 2002.

206. Roman, Manuel, Christopher K. Hess, Anand Ranganathan, Prdeep Madhavarapu, Bhaskar Borthakur, Prashant Viswanathan, Renato Cerqueira, Roy H. Campbell, and M. Dennis Mickunas, *GaiaOS: An Infrastructure for Active Spaces*. Urbana, Department of Computer Science, University of Illinois, Urbana Champaign UIUCDCS-R-2001-2224, Urbana, 2001.

207. Ryan, Nick, Jason Pascoe, and David R. Morse. *FieldNote: a Handheld Information System for the Field*. In *First International Workshop on TeloGeoProcessing (Telegeo'99)*, 1999.

208. Salutation Consortium, *Salutation Architecture Specification (Part-1) Version 2.1*. The Salutation Consortium, 1999. http://www.salutation.org

209. Salutation Consortium, *Salutation Architecture Specification (Part-2)*. The Salutation Consortium, 1999. http://www.salutation.org

210. Schatz, Bruce, *Information Retrieval in Digital Libraries: Bringing Search to the Net*. Science, *275:327-334*, 1997.

211. Schild, K. *A correspondence theory for terminlological logics: Preliminary report*. In *International Joint Conference on Artificial Intelligence*, 1991.

212. Schilit, William N., *A System Architecture for Context-Aware Mobile Computing*, in *Computer Science*. 1995, Columbia University: New York.

213. Schwidder, Jens, Tara Talbott, and James Myers. *Bootstrapping to a Semantic Grid*. In *Proceedings of the Semantic Infrastructure for Grid Computing Applications Workshop (SIGAW), at IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID)*, 2005.

214. Semantic Web Agents Project. *Pellet OWL Reasoner*. 2003, http://www.mindswap.org/2003/pellet.

215. Smith, Barry. *Ontology and Information Systems*. 2003, http://ontology.buffalo.edu/smith/articles/ontologies.htm.

216. Staab, Steffen, *Where Are the Rules?* IEEE Intelligent Systems, *18 (5):76-83*, 2003.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

217. Stuckenschmidt, Heiner. *Using OIL for Intelligent Information Integration*. In *Workshop on Applications of Ontologies and Problem-Solving Methods at ECAI*, 2000.

218. Stuckenschmidt, Heiner, Frank van Harmelen, Dieter Fensel, Michel Klein, and Ian Horrocks, *Catalogue Integration: A Case Study in Ontology-Based Semantic Translation*. 2000. http://www.ontoknowledge.org/oil/downl/CatIntegr.pdf

219. Sun Microsystems Inc., *A Collection of Jini (TM) Technology Helper Utilities and Services Specifications: Version 1.2*. 2001. xxx

220. Sun Microsystems Inc., *JINI (TM) Technology Core Platform Specification: Version 1.2*. 2001.

221. Sure, York, Juergen Angele, and Steffan Staab, *OntoEdit: Multifaceted Inferencing for Ontology Engineering*. Journal on Data Semantics, *1 (1):128-152*, 2003.

222. Sycara, Katya, Matthias Klusch, Seth Widoff, and Jianguo Lu, *Dynamic Service Matchmaking Among Agents in Open Information Environments*. ACM SIGMOD Record, *28 (1):47-53*, 1999.

223. Tablado, A., Arantza Illaramendi, J. Bermúdez, and Alfredo Goñi. *Intelligent Monitoring of Elderly People*. In *Fourth Annual IEEE EMBS Conference on Information Technology Applications in Biomedicine*, 2003, 78-81.

224. Talbott, Tara, Michael Peterson, Jens Schwidder, and James D. Myers. *Adapting the Electronic Laboratory Notebook for the Semantic Era*. In *International Symposium on Collaborative Technologies and Systems (CTS 2005)*, 2005.

225. Taligent Inc., *PEOLE PLACES AND THINGS*. 1995, Trademark No. 74360159: United States.

226. Tamma, Valentina, Michael Wooldridge, and Ian Dickinson. *An ontology based approach to automated negotiation*. In *Proceedings of the IV workshop on agent mediated electronic commerce (AMEC IV)*, 2002.

227. Tatarinov, Igor and Alon Y. Halevy. *Efficient Query Reformulation in Peer Data Management Systems*. In *SIGMOD 2004*, 2004, 539-550.

228. The Dublin Core Metadata Initiative. *Dublin Core Metadata Element Set, Version 1.1: Reference Description*. http://purl.oclc.org/docs/core/documents/rec-dces-19990702.htm.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

229. Thomson, Graham, Matthew Rithmond, Sotiros Terzis, and Paddy Nixon. *An Approach to Dynamic Context Discovery and Composition.* In *Ubisys workshop at UbiComp 2003*, 2003.

230. Toivonen, Santtu, Juha Kolari, and Timo Laakko. *Facilitating Mobile Users with Contextualized Content.* In *Artificial Intelligence in Mobile Systems Workshop at UbiComp 2003*, 2003.

231. Trastour, David, Claudio Bartolini, and Javier Gonzalez-Castillo, *A Semantic Web Approach to Service Description for Matchmaking of Services*. Bristol, HP Laboratories Bristol HPL-2001-183, Bristol, 2001. http://www.hpl.hp.com/techreports

232. Tuecke, S., K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maquire, T. Sandholm, D. Snelling, and P. Vanderbilt, *Open Grid Service Infrastructure (OGSI).* Global Grid Forum Proposed Recommendation GFD-R-P.15, 2003.

233. Turing, Alan M., *On Computable Numbers, with an Application to the Entscheidungs Problem.* Proceedings of the London Mathematical Society, *2 (42):230-265*, 1936.

234. U. S. ISBN Agency. *ISBN - The International Standard Book Number.* http://www.isgn.org/standards/home/isgn/us/.

235. uddi.org, *UDDI Technical White Paper.* 2002. http://www.uddi.org/pubs/UDDI_Executive_White_Paper.pdf

236. Ullman, Jeffrey D., *Principles of Database Systems*, Rockville, Computer Science Press, 1982.

237. UMBC. *F-OWL: An OWL inference Engine in FLORA-2.* 2003, http://fowl.sourceforge.net/about.html.

238. Van Laerhoven, Kristof and Ozan Cakmacki. *What shall we teach our pants?* In *Fourth International Symposium on Wearable Computers*, 2000, 77-83.

239. W3C. *Extensible Markup Language (XML).* http://www.w3.org/XML.

240. W3C. *Resource Description Framework (RDF).* http://www.w3c.org/RDF.

241. W3C. *XML Schema.* http://www.w3.org/XML/Schema.html.

242. W3C. *Namespaces in XML.* 1999, http://www.w3.org/TR/REC-xml-names/.

243. W3C. *Resource Description Framework (RDF) Model and Syntax Specification.* 1999, http://www.w3.org/TR/REC-rdf-syntax/.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

244. W3C. *Resource Description Framework (RDF) Schema Specification 1.0.* 2000, http://www.w3.org/TR/2000/CR-rdf-schema-20000327.

245. W3C, *XML Schema Part 2: Datatypes*. W3C W3C Recommendation, 2001. http://www.w3.org/TR/xmlschema-2/

246. W3C, *Feature Synopsis for OWL Lite and OWL.* W3C Working Draft, 2002. http://www.w3.org/TR/2002/WD-owl-features-20020729/

247. W3C. *Requirements for a Web Ontology Language.* 2002, http://www.w3c.org/TR/webont-req/.

248. W3C, *SOAP Version 1.2 Part 1: Messaging Framework.* W3C Candidate Recommendation, 2002. http://www.w3.org/TR/soap12-part1/

249. W3C, *Web Ontology Language (OWL) Guide Version 1.0.* W3C W3C Working Draft, 2002. http://www.w3.org/TR/2002/WD-owl-guide-20021104/

250. W3C, *Web Services Architecture.* W3C Working Draft, 2002. http://www.w3.org/TR/2002/WD-ws-arch-20021114/

251. W3C. *The Semantic Web.* 2003, http://www.w3.org/2001/sw.

252. W3C, *Web Ontology Language (OWL) Reference Version 1.0.* W3C Working Draft, 2003. http://www.w3.org/TR/2002/WD-owl-ref-20030331/

253. W3C, *Web Services Description Language (WSDL) Version 1.2.* W3C Working Draft, 2003. http://www.w3.org/TR/wsdl12/

254. Wagner, Gerd, *Foundations of Knowledge Systems with Applications to Databases and Agents*, Boston, Kluwer Academic Publishers, 1998.

255. Wahl, M., T. Howes, and S. Kille, *Lightweight Directory Access Protocol (v3).* IETF RFC 2251, 1997. http://www.rfc-editor.org/rfc/rfc2251.txt

256. Want, Roy and Bill Schilit, *Expanding Horizons of Location-Aware Computing.* IEEE Computer, *34 (8):31-34*, 2001.

257. Weiser, Mark, *The computer in the 21st century.* Scientific American, *265 (3):66-75*, 1991.

258. Wichadakul, Duangdao, Xiaohui Gu, and Klara Nahrstedt. *A Programming Framework for Quality-Aware Ubiquitous Multimedia Applications.* In *ACM Multimedia*, 2002.

259. Widom, Jennifer. *Research Problems in Data Warehousing.* In *International Conference on Information and Knowledge Management*, 1995.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

260. Winograd, Terry, *Architectures for Context.* Human-Computer Interaction, *16 (in press)* 2001.

261. Wu, Dan, Bijan Parsia, Evren Sirin, James Hendler, and Dana Nau. *Automating DAML-S Web Services Composition Using SHOP2.* In *Second International Semantic Web Conference*, 2003, 195-210.

262. Yeager, Nancy J. and Robert E. McGrath, *Web Server Technology: The Authoritative Guide*, San Francisco, Morgan-Kaufmann, 1996.

263. Zhang, Dell and Wee Sun Lee, *Learning to Integrate Web Taxonomies.* Web Semantics, *2 (2):131-151*, December 15 2004.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Appendix 1: Listings

Listing 1

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <rdf:RDF xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
3    xmlns:dc="http://purl.org/dc/elements/1.1/"
4    xmlns:OilEd="http://img.cs.man.ac.uk/oil/OilEd#"
5    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
6    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#" xmlns:xsd="http://www.w3.org/2000/10/XMLSchema#">
7    <daml:Ontology rdf:about="">
8       <dc:title>&quot;An Ontology&quot;</dc:title>
9       <dc:date></dc:date>
10      <dc:creator></dc:creator>
11      <dc:description></dc:description>
12      <dc:subject></dc:subject>
13      <daml:versionInfo></daml:versionInfo>
14   </daml:Ontology>
15   <daml:Class rdf:about="http://somewhere.net/sd.daml#Memory">
16   </daml:Class>
17   <daml:Class rdf:about="http://somewhere.net/sd.daml#SERV11">
18      <rdfs:subClassOf>
19         <daml:Class rdf:about="http://somewhere.net/sd.daml#CD9900C1"/>
20      </rdfs:subClassOf>
21   </daml:Class>
22   <daml:Class rdf:about="http://somewhere.net/sd.daml#SERV12">
23      <rdfs:subClassOf>
24         <daml:Class rdf:about="http://somewhere.net/sd.daml#CD9605I"/>
25      </rdfs:subClassOf>
26   </daml:Class>
27   <daml:Class rdf:about="http://somewhere.net/sd.daml#CD9900C1">
28      <rdfs:subClassOf>
29         <daml:Class rdf:about="http://somewhere.net/sd.daml#CD"/>
30      </rdfs:subClassOf>
31   </daml:Class>
32   <daml:Class rdf:about="http://somewhere.net/sd.daml#Service-Description">
33   </daml:Class>
34   <daml:Class rdf:about="http://somewhere.net/sd.daml#computer">
35      <rdfs:subClassOf>
36         <daml:Class rdf:about="http://somewhere.net/sd.daml#Service-Description"/>
37      </rdfs:subClassOf>
38      <rdfs:subClassOf>
39         <daml:Restriction>
40            <daml:onProperty rdf:resource="http://somewhere.net/sd.daml#hasMemory"/>
41            <daml:hasClass>
42               <daml:Class rdf:about="http://somewhere.net/sd.daml#Memory"/>
43            </daml:hasClass>
44         </daml:Restriction>
45      </rdfs:subClassOf>
46      <rdfs:subClassOf>
47         <daml:Restriction>
48            <daml:onProperty rdf:resource="http://somewhere.net/sd.daml#hasPrinter"/>
49            <daml:hasClass rdf:resource="http://www.w3.org/2000/10/XMLSchema#boolean"/>
50         </daml:Restriction>
51      </rdfs:subClassOf>
52      <rdfs:subClassOf>
53         <daml:Restriction>
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
54            <daml:onProperty rdf:resource="http://somewhere.net/sd.daml#hasDVD"/>
55            <daml:hasClass>
56               <daml:Class rdf:about="http://somewhere.net/sd.daml#DVD"/>
57            </daml:hasClass>
58         </daml:Restriction>
59      </rdfs:subClassOf>
60      <rdfs:subClassOf>
61         <daml:Restriction>
62            <daml:onProperty rdf:resource="http://somewhere.net/sd.daml#hasDVD"/>
63            <daml:hasClass>
64               <daml:Class rdf:about="http://somewhere.net/sd.daml#DVD"/>
65            </daml:hasClass>
66         </daml:Restriction>
67      </rdfs:subClassOf>
68   </daml:Class>
69   <daml:Class rdf:about="http://somewhere.net/sd.daml#OVER256">
70      <rdfs:subClassOf>
71         <daml:Class rdf:about="http://somewhere.net/sd.daml#Memory"/>
72      </rdfs:subClassOf>
73   </daml:Class>
74   <daml:Class rdf:about="http://somewhere.net/sd.daml#CD">
75      <rdfs:subClassOf>
76         <daml:Class rdf:about="http://somewhere.net/sd.daml#Service-Description"/>
77      </rdfs:subClassOf>
78   </daml:Class>
79   <daml:Class rdf:about="http://somewhere.net/sd.daml#DVD">
80      <rdfs:subClassOf>
81         <daml:Class rdf:about="http://somewhere.net/sd.daml#Service-Description"/>
82      </rdfs:subClassOf>
83   </daml:Class>
84   <daml:Class rdf:about="http://somewhere.net/sd.daml#SERV1">
85      <rdfs:subClassOf>
86         <daml:Class rdf:about="http://somewhere.net/sd.daml#SERV4"/>
87      </rdfs:subClassOf>
88      <rdfs:subClassOf>
89         <daml:Class rdf:about="http://somewhere.net/sd.daml#SERV6"/>
90      </rdfs:subClassOf>
91      <rdfs:subClassOf>
92         <daml:Restriction daml:cardinalityQ="1">
93            <daml:onProperty rdf:resource="http://somewhere.net/sd.daml#hasMemory"/>
94            <daml:hasClassQ>
95               <daml:Class rdf:about="http://somewhere.net/sd.daml#OVER256"/>
96            </daml:hasClassQ>
97         </daml:Restriction>
98      </rdfs:subClassOf>
99      <rdfs:subClassOf>
100        <daml:Restriction daml:cardinalityQ="1">
101           <daml:onProperty rdf:resource="http://somewhere.net/sd.daml#hasPrinter"/>
102           <daml:hasClassQ rdf:resource="http://www.w3.org/2000/10/XMLSchema#boolean"/>
103        </daml:Restriction>
104     </rdfs:subClassOf>
105     <rdfs:subClassOf>
106        <daml:Restriction daml:cardinalityQ="1">
107           <daml:onProperty rdf:resource="http://somewhere.net/sd.daml#hasDVD"/>
108           <daml:hasClassQ>
109              <daml:Class rdf:about="http://somewhere.net/sd.daml#DVDxxx"/>
110           </daml:hasClassQ>
111        </daml:Restriction>
112     </rdfs:subClassOf>
113     <rdfs:subClassOf>
114        <daml:Restriction>
115           <daml:onProperty rdf:resource="http://somewhere.net/sd.daml#hasCD"/>
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
116             <daml:hasClass>
117                <daml:Class rdf:about="http://somewhere.net/sd.daml#CD9605I"/>
118             </daml:hasClass>
119          </daml:Restriction>
120       </rdfs:subClassOf>
121    </daml:Class>
122    <daml:Class rdf:about="http://somewhere.net/sd.daml#SERV2">
123       <rdfs:subClassOf>
124          <daml:Class rdf:about="http://somewhere.net/sd.daml#computer"/>
125       </rdfs:subClassOf>
126       <rdfs:subClassOf>
127          <daml:Restriction daml:cardinalityQ="1">
128             <daml:onProperty rdf:resource="http://somewhere.net/sd.daml#hasMemory"/>
129             <daml:hasClassQ>
130                <daml:Class rdf:about="http://somewhere.net/sd.daml#OVER256"/>
131             </daml:hasClassQ>
132          </daml:Restriction>
133       </rdfs:subClassOf>
134    </daml:Class>
135    <daml:Class rdf:about="http://somewhere.net/sd.daml#SERV3">
136       <rdfs:subClassOf>
137          <daml:Class rdf:about="http://somewhere.net/sd.daml#SERV2"/>
138       </rdfs:subClassOf>
139       <rdfs:subClassOf>
140          <daml:Restriction daml:cardinalityQ="1">
141             <daml:onProperty rdf:resource="http://somewhere.net/sd.daml#hasPrinter"/>
142             <daml:hasClassQ rdf:resource="http://www.w3.org/2000/10/XMLSchema#string"/>
143          </daml:Restriction>
144       </rdfs:subClassOf>
145    </daml:Class>
146    <daml:Class rdf:about="http://somewhere.net/sd.daml#SERV4">
147       <rdfs:subClassOf>
148          <daml:Class rdf:about="http://somewhere.net/sd.daml#SERV5"/>
149       </rdfs:subClassOf>
150       <rdfs:subClassOf>
151          <daml:Restriction daml:cardinalityQ="1">
152             <daml:onProperty rdf:resource="http://somewhere.net/sd.daml#hasPrinter"/>
153             <daml:hasClassQ rdf:resource="http://www.w3.org/2000/10/XMLSchema#boolean"/>
154          </daml:Restriction>
155       </rdfs:subClassOf>
156       <rdfs:subClassOf>
157          <daml:Restriction daml:cardinalityQ="1">
158             <daml:onProperty rdf:resource="http://somewhere.net/sd.daml#hasDVD"/>
159             <daml:hasClassQ>
160                <daml:Class rdf:about="http://somewhere.net/sd.daml#DVDxxx"/>
161             </daml:hasClassQ>
162          </daml:Restriction>
163       </rdfs:subClassOf>
164       <rdfs:subClassOf>
165          <daml:Restriction>
166             <daml:onProperty rdf:resource="http://somewhere.net/sd.daml#hasCD"/>
167             <daml:hasClass>
168                <daml:Class rdf:about="http://somewhere.net/sd.daml#CD9605I"/>
169             </daml:hasClass>
170          </daml:Restriction>
171       </rdfs:subClassOf>
172    </daml:Class>
173    <daml:Class rdf:about="http://somewhere.net/sd.daml#SERV5">
174       <rdfs:subClassOf>
175          <daml:Class rdf:about="http://somewhere.net/sd.daml#SERV9"/>
176       </rdfs:subClassOf>
177       <rdfs:subClassOf>
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
178         <daml:Restriction>
179            <daml:onProperty rdf:resource="http://somewhere.net/sd.daml#hasCD"/>
180            <daml:hasClass>
181               <daml:Class rdf:about="http://somewhere.net/sd.daml#CD96051"/>
182            </daml:hasClass>
183         </daml:Restriction>
184      </rdfs:subClassOf>
185   </daml:Class>
186   <daml:Class rdf:about="http://somewhere.net/sd.daml#SERV6">
187      <rdfs:subClassOf>
188         <daml:Class rdf:about="http://somewhere.net/sd.daml#SERV7"/>
189      </rdfs:subClassOf>
190      <rdfs:subClassOf>
191         <daml:Restriction daml:cardinalityQ="1">
192            <daml:onProperty rdf:resource="http://somewhere.net/sd.daml#hasPrinter"/>
193            <daml:hasClassQ rdf:resource="http://www.w3.org/2000/10/XMLSchema#boolean"/>
194         </daml:Restriction>
195      </rdfs:subClassOf>
196      <rdfs:subClassOf>
197         <daml:Restriction daml:cardinalityQ="1">
198            <daml:onProperty rdf:resource="http://somewhere.net/sd.daml#hasMemory"/>
199            <daml:hasClassQ>
200               <daml:Class rdf:about="http://somewhere.net/sd.daml#OVER256"/>
201            </daml:hasClassQ>
202         </daml:Restriction>
203      </rdfs:subClassOf>
204      <rdfs:subClassOf>
205         <daml:Restriction daml:cardinalityQ="1">
206            <daml:onProperty rdf:resource="http://somewhere.net/sd.daml#hasCD"/>
207            <daml:hasClassQ>
208               <daml:Class rdf:about="http://somewhere.net/sd.daml#CD"/>
209            </daml:hasClassQ>
210         </daml:Restriction>
211      </rdfs:subClassOf>
212      <rdfs:subClassOf>
213         <daml:Restriction daml:cardinalityQ="1">
214            <daml:onProperty rdf:resource="http://somewhere.net/sd.daml#hasDVD"/>
215            <daml:hasClassQ>
216               <daml:Class rdf:about="http://somewhere.net/sd.daml#DVDxxx"/>
217            </daml:hasClassQ>
218         </daml:Restriction>
219      </rdfs:subClassOf>
220   </daml:Class>
221   <daml:Class rdf:about="http://somewhere.net/sd.daml#SERV7">
222      <rdfs:subClassOf>
223         <daml:Class rdf:about="http://somewhere.net/sd.daml#SERV3"/>
224      </rdfs:subClassOf>
225      <rdfs:subClassOf>
226         <daml:Class rdf:about="http://somewhere.net/sd.daml#SERV9"/>
227      </rdfs:subClassOf>
228      <rdfs:subClassOf>
229         <daml:Restriction daml:cardinalityQ="1">
230            <daml:onProperty rdf:resource="http://somewhere.net/sd.daml#hasPrinter"/>
231            <daml:hasClassQ rdf:resource="http://www.w3.org/2000/10/XMLSchema#boolean"/>
232         </daml:Restriction>
233      </rdfs:subClassOf>
234      <rdfs:subClassOf>
235         <daml:Restriction daml:cardinalityQ="1">
236            <daml:onProperty rdf:resource="http://somewhere.net/sd.daml#hasMemory"/>
237            <daml:hasClassQ>
238               <daml:Class rdf:about="http://somewhere.net/sd.daml#OVER256"/>
239            </daml:hasClassQ>
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
240         </daml:Restriction>
241       </rdfs:subClassOf>
242       <rdfs:subClassOf>
243         <daml:Restriction daml:cardinalityQ="1">
244           <daml:onProperty rdf:resource="http://somewhere.net/sd.daml#hasCD"/>
245           <daml:hasClassQ>
246             <daml:Class rdf:about="http://somewhere.net/sd.daml#CD"/>
247           </daml:hasClassQ>
248         </daml:Restriction>
249       </rdfs:subClassOf>
250     </daml:Class>
251     <daml:Class rdf:about="http://somewhere.net/sd.daml#SERV8">
252       <rdfs:subClassOf>
253         <daml:Class rdf:about="http://somewhere.net/sd.daml#SERV9"/>
254       </rdfs:subClassOf>
255       <rdfs:subClassOf>
256         <daml:Restriction>
257           <daml:onProperty rdf:resource="http://somewhere.net/sd.daml#hasCD"/>
258           <daml:hasClass>
259             <daml:Class rdf:about="http://somewhere.net/sd.daml#CD9900C1"/>
260           </daml:hasClass>
261         </daml:Restriction>
262       </rdfs:subClassOf>
263     </daml:Class>
264     <daml:Class rdf:about="http://somewhere.net/sd.daml#SERV9">
265       <rdfs:subClassOf>
266         <daml:Class rdf:about="http://somewhere.net/sd.daml#computer"/>
267       </rdfs:subClassOf>
268       <rdfs:subClassOf>
269         <daml:Restriction daml:cardinalityQ="1">
270           <daml:onProperty rdf:resource="http://somewhere.net/sd.daml#hasCD"/>
271           <daml:hasClassQ>
272             <daml:Class rdf:about="http://somewhere.net/sd.daml#CD"/>
273           </daml:hasClassQ>
274         </daml:Restriction>
275       </rdfs:subClassOf>
276     </daml:Class>
277     <daml:Class rdf:about="http://somewhere.net/sd.daml#CD9605I">
278       <rdfs:subClassOf>
279         <daml:Class rdf:about="http://somewhere.net/sd.daml#CD"/>
280       </rdfs:subClassOf>
281     </daml:Class>
282     <daml:Class rdf:about="http://somewhere.net/sd.daml#DVDxxx">
283       <rdfs:subClassOf>
284         <daml:Class rdf:about="http://somewhere.net/sd.daml#DVD"/>
285       </rdfs:subClassOf>
286     </daml:Class>
287     <daml:ObjectProperty rdf:about="http://somewhere.net/sd.daml#hasDVD">
288       <rdfs:domain>
289         <daml:Class rdf:about="http://somewhere.net/sd.daml#computer"/>
290       </rdfs:domain>
291       <rdfs:range>
292         <daml:Class rdf:about="http://somewhere.net/sd.daml#DVD"/>
293       </rdfs:range>
294     </daml:ObjectProperty>
295     <daml:ObjectProperty rdf:about="http://somewhere.net/sd.daml#hasMemory">
296       <rdfs:domain>
297         <daml:Class rdf:about="http://somewhere.net/sd.daml#computer"/>
298       </rdfs:domain>
299       <rdfs:range>
300         <daml:Class rdf:about="http://somewhere.net/sd.daml#Memory"/>
301       </rdfs:range>
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
302 </daml:ObjectProperty>
303 <daml:ObjectProperty rdf:about="http://somewhere.net/sd.daml#hasCD">
304   <rdfs:domain>
305     <daml:Class rdf:about="http://somewhere.net/sd.daml#computer"/>
306   </rdfs:domain>
307   <rdfs:range>
308     <daml:Class rdf:about="http://somewhere.net/sd.daml#CD"/>
309   </rdfs:range>
310 </daml:ObjectProperty>
311 <daml:DatatypeProperty rdf:about="http://somewhere.net/sd.daml#hasPrinter">
312   <rdfs:domain>
313     <daml:Class rdf:about="http://somewhere.net/sd.daml#computer"/>
314   </rdfs:domain>
315   <rdfs:range>
316     <xsd:boolean/>
317   </rdfs:range>
318 </daml:DatatypeProperty>
319 <daml:Class rdf:about="http://somewhere.net/sd.daml#CD9605I">
320   <daml:disjointWith>
321     <daml:Class rdf:about="http://somewhere.net/sd.daml#CD9900C1"/>
322   </daml:disjointWith>
323 </daml:Class>
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Listing 2

```
 1 <daml:Class rdf:about="file:C:/ActiveSpaces/Semantics/MyOntology/ActiveSpace.daml#MP3Server">
 2  <rdfs:label>MP3Server</rdfs:label>
 3  <rdfs:comment><![CDATA[An MP3Server maintains a list of songs - this list can be searched by certain attributes and it can also be sent
commands to play songs]]></rdfs:comment>
 4  <OilEd:creationDate><![CDATA[2002-11-09T17:10:52Z]]></OilEd:creationDate>
 5  <OilEd:creator><![CDATA[ranganat]]></OilEd:creator>
 6  <rdfs:subClassOf>
 7   <daml:Class rdf:about="file:C:/ActiveSpaces/Semantics/MyOntology/ActiveSpace.daml#SearchableService"/>
 8  </rdfs:subClassOf>
 9  <rdfs:subClassOf>
10   <daml:Class rdf:about="file:C:/ActiveSpaces/Semantics/MyOntology/ActiveSpace.daml#CommandableService"/>
11  </rdfs:subClassOf>
12  <rdfs:subClassOf>
13   <daml:Restriction>
14    <daml:onProperty rdf:resource="file:C:/ActiveSpaces/Semantics/MyOntology/ActiveSpace.daml#executesDataType"/>
15    <daml:hasClass>
16     <daml:Class rdf:about="file:C:/ActiveSpaces/Semantics/MyOntology/ActiveSpace.daml#MP3File"/>
17    </daml:hasClass>
18   </daml:Restriction>
19  </rdfs:subClassOf>
20  <rdfs:subClassOf>
21   <daml:Restriction>
22    <daml:onProperty rdf:resource="file:C:/ActiveSpaces/Semantics/MyOntology/ActiveSpace.daml#searchableBy"/>
23    <daml:hasClass>
24     <daml:Class rdf:about="file:C:/ActiveSpaces/Semantics/MyOntology/ActiveSpace.daml#MP3Attributes"/>
25    </daml:hasClass>
26   </daml:Restriction>
27  </rdfs:subClassOf>
28  <rdfs:subClassOf>
29   <daml:Restriction>
30    <daml:onProperty rdf:resource="file:C:/ActiveSpaces/Semantics/MyOntology/ActiveSpace.daml#commandableBy"/>
31    <daml:hasClass>
32     <daml:Class rdf:about="file:C:/ActiveSpaces/Semantics/MyOntology/ActiveSpace.daml#MP3ServerPlay"/>
33    </daml:hasClass>
34   </daml:Restriction>
35  </rdfs:subClassOf>
36  <rdfs:subClassOf>
37   <daml:Restriction>
38    <daml:onProperty rdf:resource="file:C:/ActiveSpaces/Semantics/MyOntology/ActiveSpace.daml#commandableBy"/>
39    <daml:hasClass>
40     <daml:Class rdf:about="file:C:/ActiveSpaces/Semantics/MyOntology/ActiveSpace.daml#MP3ServerRandomPlay"/>
41    </daml:hasClass>
42   </daml:Restriction>
43  </rdfs:subClassOf>
44 </daml:Class
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Listing 3

```
1 <daml:Class rdf:about="file:C:/ActiveSpaces/Semantics/MyOntology/ActiveSpace.daml#Temperature">
2  <rdfs:label>TemperatureInformation</rdfs:label>
3  <rdfs:comment><![CDATA[]]></rdfs:comment>
4  <OilEd:creationDate><![CDATA[2002-10-06T19:18:06Z]]></OilEd:creationDate>
5  <OilEd:creator><![CDATA[ranganat]]></OilEd:creator>
6  <rdfs:subClassOf>
7   <daml:Class rdf:about="file:C:/ActiveSpaces/Semantics/MyOntology/ActiveSpace.daml#WeatherInformation"/>
8  </rdfs:subClassOf>
9  <rdfs:subClassOf>
10   <daml:Restriction daml:cardinalityQ="1">
11    <daml:onProperty rdf:resource="file:C:/ActiveSpaces/Semantics/MyOntology/ActiveSpace.daml#subject"/>
12    <daml:hasClassQ>
13     <daml:Class>
14      <daml:unionOf>
15       <daml:List>
16        <daml:first>
17          <daml:Class rdf:about="file:C:/ActiveSpaces/Semantics/MyOntology/ActiveSpace.daml#PhysicalPlace"/>
18        </daml:first>
19        <daml:rest>
20         <daml:List>
21         <daml:first>
22          <daml:Class rdf:about="file:C:/ActiveSpaces/Semantics/MyOntology/ActiveSpace.daml#Person"/>
23         </daml:first>
24         <daml:rest>
25          <daml:nil/>
26         </daml:rest>
27         </daml:List>
28         </daml:rest>
29        </daml:List>
30      </daml:unionOf>
31     </daml:Class>
32    </daml:hasClassQ>
33   </daml:Restriction>
34  </rdfs:subClassOf>
35  <rdfs:subClassOf>
36   <daml:Restriction daml:cardinalityQ="1">
37    <daml:onProperty rdf:resource="file:C:/ActiveSpaces/Semantics/MyOntology/ActiveSpace.daml#relator"/>
38    <daml:hasClassQ>
39      <daml:Class rdf:about="file:C:/ActiveSpaces/Semantics/MyOntology/ActiveSpace.daml#ComparisonOperator"/>
40    </daml:hasClassQ>
41   </daml:Restriction>
42  </rdfs:subClassOf>
43  <rdfs:subClassOf>
44   <daml:Restriction daml:cardinalityQ="1">
45    <daml:onProperty rdf:resource="file:C:/ActiveSpaces/Semantics/MyOntology/ActiveSpace.daml#object"/>
46    <daml:hasClassQ>
47      <daml:Class rdf:about="file:C:/ActiveSpaces/Semantics/MyOntology/ActiveSpace.daml#TemperatureValue"/>
48    </daml:hasClassQ>
49   </daml:Restriction>
50  </rdfs:subClassOf>
51  </daml:Class>
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Listing 4

```
1 // This is the CORBA IDL for the FaCT server
2 //
3 //  This IDL was taken from the published report:
4 //
5 //    Sean Bechofer, Ian Horrocks, and Sergio Tessaris, "CORBA Interface
6 //      for a DL Classifier".  Available as part of the FaCT distribution
7 //      in the file manual/corba-fact-idl.pdf
8 //
9
10 //
11 //  This IDL is used to generate client stubs when needed.  The server
12 //  is not recompiled.
13 //
14
15
16 module img {
17 module fact {
18    typedef string ConceptDescription;
19    typedef string RoleDescription;
20    typedef string cName;
21    typedef string rName;
22    typedef sequence <cName> cEquivNames;
23    typedef sequence <cEquivNames> cNames;
24    typedef sequence <rName> rEquivNames;
25    typedef sequence <rEquivNames> rNames;
26
27    struct taxonomyNode {
28       cNames supers;
29       cNames subs;
30       cEquivNames equivs;
31    };
32
33    exception CommunicationException {
34       short code;
35       string information;
36    };
37    exception TransactionRequiredException {
38       short code;
39       string information;
40    };
41    exception OpUnimplementedException {
42       short code;
43       string information;
44    };
45    exception ExprErrorException {
46       short code;
47       string information;
48    };
49    exception KBModifiedException {
50       short code;
51       string information;
52 };
53
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
54  interface ClientHandler {
55
56    string identifier();
57    void release();
58
59    boolean begin_transaction();
60    boolean end_transaction();
61    boolean abort_transaction();
62    boolean in_transaction();
63
64    //  'tells': assertions to KB
65
66    void clear()
67      raises (TransactionRequiredException, OpUnimplementedException,
68        ExprErrorException);
69    void defconcept(in cName nm)
70      raises (TransactionRequiredException, OpUnimplementedException,
71        ExprErrorException);
72    void defrole(in rName nm)
73      raises (TransactionRequiredException, OpUnimplementedException,
74        ExprErrorException);
75    void impliesC(in ConceptDescription c1, in ConceptDescription c2)
76      raises (TransactionRequiredException, OpUnimplementedException,
77        ExprErrorException);
78    void equalC(in ConceptDescription c1, in ConceptDescription c2)
79      raises (TransactionRequiredException, OpUnimplementedException,
80        ExprErrorException);
81    void impliesR(in RoleDescription r1, in RoleDescription r2)
82      raises (TransactionRequiredException, OpUnimplementedException,
83        ExprErrorException);
84    void equalR(in RoleDescription r1, in RoleDescription r2)
85      raises (TransactionRequiredException, OpUnimplementedException,
86        ExprErrorException);
87    void transitive(in rName rn)
88      raises (TransactionRequiredException, OpUnimplementedException,
89        ExprErrorException);
90    void functional(in rName rn)
91      raises (TransactionRequiredException, OpUnimplementedException,
92        ExprErrorException);
93    string tells();
94    string allTells();
95
96    // 'asks': queries about the KB
97
98    boolean satisfiable(in ConceptDescription c)
99      raises (OpUnimplementedException, ExprErrorException,
100       KBModifiedException);
101    boolean subsumes(in ConceptDescription c1, in ConceptDescription c2)
102     raises (OpUnimplementedException, ExprErrorException,
103       KBModifiedException);
104
105    boolean equivalent(in ConceptDescription c1, in ConceptDescription c2)
106     raises (OpUnimplementedException, ExprErrorException,
107       KBModifiedException);
108
109    cNames directSupersC(in cName cn)
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
110       raises (OpUnimplementedException, ExprErrorException,
111         KBModifiedException);
112     cNames allSupersC(in cName cn)
113       raises (OpUnimplementedException, ExprErrorException,
114         KBModifiedException);
115     cNames directSubsC(in cName cn)
116      raises (OpUnimplementedException, ExprErrorException,
117         KBModifiedException);
118     cNames allSubsC(in cName cn)
119       raises (OpUnimplementedException, ExprErrorException,
120         KBModifiedException);
121     rNames directSupersR(in rName rn)
122       raises (OpUnimplementedException, ExprErrorException,
123         KBModifiedException);
124     rNames allSupersR(in rName rn)
125       raises (OpUnimplementedException, ExprErrorException,
126         KBModifiedException);
127     rNames directSubsR(in rName rn)
128       raises (OpUnimplementedException, ExprErrorException,
129         KBModifiedException);
130     rNames allSubsR(in rName rn)
131       raises (OpUnimplementedException, ExprErrorException,
132         KBModifiedException);
133     taxonomyNode taxonomyPosition(in ConceptDescription c)
134       raises (OpUnimplementedException, ExprErrorException,
135         KBModifiedException);
136   };
137
138   interface Classifier {
139     ClientHandler newHandler(in string id)
140       raises (CommunicationException);
141
142     string identifier();
143   };
144 };
145 };
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Vita

Robert E. McGrath was born in Ann Arbor Michigan in 1954. He completed a BS in Anthropology and Psychology at University of Illinois, Urbana-Champaign in 1976, and an MS in Psychology from the University of New Hampshire in 1980. He completed his MCS at the University of Illinois at Urbana-Champaign in 1985.

He has worked as a software engineer for more than 20 years in several departments at University of Illinois, Urbana-Champaign as well as private companies. Since 1994, he has been a senior software developer and team leader at the National Center for Supercomputing Applications (NCSA), at the University of Illinois at Urbana-Champaign.

He has published dozens of reports and papers on distributed systems, scientific data systems, digital libraries, and advanced information technologies. At NCSA, he was a pioneer of World Wide Web, and co-authored (with Nancy Yeager) the book, *Web Server Technology: the Authoritative Guide,* (Morgan Kaufmann, 1996). He was Principle Investigator of the project "Accessing Space Science Data Using the Internet," funded by the NASA Applied Information Research Program (NRA-96-OSS-10, 1997-1999).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.