A Service for Aggregating and Interpreting Contextual Information

Filip Perich
Software Technology Laboratory
HP Laboratories Palo Alto
HPL-2002-241
August 26th , 2002*

E-mail: filip.perich@hp.com

context-aware computing, Me-Centric computing, ontology, reasoning, web services, semantic web

We present an architectural support for building context-aware applications. With the increasing acceptance of dynamic mobile and ubiquitous computing environments, applications are no longer limited to relying only on the information explicitly provided by users. Instead, these environments invite a new kind of applications leveraging from implicit information that can range from a simple time-of-a-day information to information about user location to even an elaborate description of user current activity. This implicit information is known as the context, and although the context-aware computing paradigm has been introduced over a decade ago, so far there exist only a limited number of context-aware applications available to everyday users. We believe that this is due to the tight coupling of contextual operations where each application is required to sense, collect, interpret and apply all information independently from others. To overcome this issue we propose a decoupled approach, where the information sensing, aggregation, and interpretation is provided by external services, and applications are concerned with only applying the contextual information to their needs. Moreover, we focus on the design and specification of a generic service capable of both aggregating and interpreting context information as part of the Me-Centric computing project, and demonstrate its capability in an office-based scenario.

* Internal Accession Date Only
Approved for External Publication

Contents

Chapter 1

Introduction

The goal of the *Me-Centric* computing project is to make appliances and web services work for people instead of people working on them. The project envisions a *me-centric* world where the user is the ruler, while appliances, infrastructure and services act as servants doing due diligence to their leader. Moreover, unlike other existing context-aware frameworks the aim of the *Me-Centric* project is not on delivering any service at any time anywhere but rather on delivering the *right* service at the *right* moment [5].

To accomplish this task, the *Me-Centric* project relies heavily on a transparent acquisition of numerous information that can be readily obtained by sensing the physical environment, and on a detection and an inference of additional knowledge from the sensed information. The project applies this explicit as well as implicit knowledge to discover the user context and to select and delegate the appropriate tasks among the appliances and services to satisfy the user needs.

We define the user context in terms of *domains*. Each *domain* represents a specific part of the currently examined environment defined in terms of properties and applicable policies. A *domain* can, for example, represent a time or a fact that person is currently at work. Consequently, *domains* permit fragmenting the world into smaller parts that can be easily managed by policies. The key advantage to the notion of a *domain* is that it provides a level of abstraction. This leverages the need of an application that is utilizing the *domain* knowledge from being overwhelmed by the logic required to determine the appropriate *domain*.

Additionally, we can distinguish among four types of domains, namely: (i) *physical domains*, (ii) *knowledge domains*, (iii) *conceptual domains*, and (iv) *collaborative domains*.

The *physical domains* represent domains that are linked to the physical world with well-defined borders. An office building is an example of such domain. The building has a well-defined border by its actual walls, and its domain membership determination is therefore straight forward.

The *knowledge domains* consist of a class of virtual domains, which memberships are determined through a computation or file system access. For example, an employee database represents a *knowledge domain* as the domain membership does not depend on any sensing but rather on a database lookup. Therefore, each *knowledge domain* represents an collection of explicit facts.

The *conceptual domains* are a level above the previous two domains, and depend on an interpretation of physical and knowledge domains. A *domain* representing a meeting is an example of a *conceptual domain* as its membership requires not only a physical sensing of the location and time but also additional reasoning over calendar and other information.

The last set of domains consist of *collaborative domains*, which depend on reasoning over knowledge of many individuals to allow services to schedule and provide tasks for more than one person at a time.

We employ these concepts of domains and their properties in designing and implementing the *Me-Centric Domain Server* ($M_{er}C_eD_eS$), a generic service for aggregating and interpreting contextual information, *i.e.* domain memberships. It serves as a focal repository point, which collects sensed information provided by sensors, converts them into *domain*-based knowledge as well as reasons over this knowledge, and makes it available for querying by other services, appliances, and infrastructures. This service thus enables us to represent real and virtual environments and to delegate tasks to other services.

In Chapter 2 we describe the impact of the *Me-Centric project*. In Chapter 3 we briefly talk about existing related work and provide some background useful for describing our approach, which we present in Chapter 4. In Chapter 5 we define the design and specification of the *Me-Centric Domain Server*, and describe the implemented architecture in Chapter 6. We illustrate the capabilities of the service in Chapter 7 by defining an office-based scenario, and we summarize with Chapter 8.

Chapter 2

Impact

We have emphasized in the previous chapter that the aim of the *Me-Centric* computing project is to provide users with the *right* service at the *right* moment. This allows any infrastructure to provide their clients with customized services that are relevant and beneficial to the client current context. This will lead into a higher rate of acceptance and usage of such infrastructure as the clients will be keen to use it more frequently, in turn leading into a higher revenue.

We can illustrate the benefits of the *Me-Centric* approach by considering an example that is loosely based on a real case scenario of a web server usage from [10]. In this example, a car-rental company gives individualized offers to each person traveling through the J.F.K. International Airport by determining the person's final destination, number of companions and other information. This is different from the existing scenario where all people are present with the same information at all times. Here, the car rental company may instead offer a more lucrative deal to Bob than Alice although they are both arriving at J.F.K. in the same plane on Friday afternoon. The car-rental company gives a better offer to Bob because it determines that Bob is ending his N.Y.C. visit on Monday morning at the Lu-Guardia Airport. While Alice, on the other hand, is more likely to return the car at J.F.K. as her returning flight is scheduled from that location. Since the car-rental company requires as many cars as possible at the Lu-Guardia Airport to satisfy all reservations made by businessmen arriving on that Monday morning, the company prefers to rent out a car

that is guaranteed to be there instead of renting the car that is returned elsewhere and having to transport another car to the required location.

The above example clearly describes a scenario, which can be easily build on top of the *Me-Centric* infrastructure. The scenario requires sensing capabilities, converting it into abstract *domain* representation, and deducing the appropriate *domain* memberships for both Alice and Bob. As the same information is most likely applicable to additional services, we can also see the benefit of providing one global service that is able to aggregate and deduce all the necessary knowledge, and that makes it available to the end-point services. This is precisely the task of the *Me-Centric Domain Server*.

The use of the *Me-Centric Domain Server* is therefore applicable and highly beneficial anytime there is a need to collect and maintain contextual information for the use and re-use by multiple services. Additionally, the use of the *Me-Centric Domain Server* allows for a rapid creation of other services as the developers must only concentrate on the service functionality and do not have to worry about sensing the necessary context, a functionality provided by the *Me-Centric Domain Server*.

Chapter 3

Background

In this chapter, we briefly define the concept *context* and challenges in aggregating and reasoning over it. Additionally, we describe some existing context-aware applications, and conclude with a brief introduction to a Semantic Mark-Ups as a mean for providing enhanced Web Services.

3.1 On Context Awareness

Dey *et al* [9] have recently provided an excellent state-of-the-art overview and introduction to problems concerned with handling of context. They define *context* in the following terms:

> **Definition** Context is any information that can be used to characterize the situation of entities (*i.e.* whether a person, place or object) that are considered relevant to the interaction between a user and an application, including the user and the application themselves. Context is typically the location, identity and state of people, groups and computational and physical objects [9, 8].

This definition implies that for an application to be truly *context-aware* it must first be able to acquire context information, and second it must be able to use it in an *intelligent* manner. By *intelligent* manner, we mean a manner that is beneficial to either the service, the user or both. For the purpose of this document, we focus on the first part. This part requires that the application is able to acquire as much of context information as possible and that is relevant to its offered services and/or to its user.

Dey *et al* define four categories of context information that is generally sought by context-aware applications, and which we can also describe in terms of *domains*. These categories consist of *identity*, *location*, *activity/status* and *time*. The *identity* and *time* categories are examples of *knowledge domains* as they require a database lookup and a calculation, respectively. The *location* category is an example of a *physical domain* as it is clearly bounded inside a well-defined physical area. Lastly, the *activity/status* category is an example of a *conceptual domain*.

From the translation into the *domain* concepts, we see that acquisition and determination of *activity/status* must be the hardest category. This is also the reason why many "context-aware applications", such as HP Cooltown [13], Agents2Go [19] or Cricket [17], are primarily concerned with identifying and locating entities, and calculating time only. Although, a preliminary acquisition of information useful for describing user activity and status have also began to be tackled recently [16].

An important challenge however remains in deciding who is responsible for collecting and reasoning over context information? Where the information should be stored? And how it should be represented to provide constant availability and reliability as well as to provide high overall scalability? Clearly, solutions requiring each application to store their own contextual information are not efficient, as many applications are likely to duplicate actions and replicate storage requirements of others. Additionally, an approach that simply stores context information in a centralized static database and allows services to query it is also inefficient. This is because the querying clients are still likely to duplicate reasoning actions of their peers.

We, therefore, require that a context-aware framework should consist of three parts. The first part is responsible for discovering physical information by sensing or via other means. Second part is responsible for storing and managing the context information, and for inferring additional context knowledge. While the third part is responsible for applying the context to customize its services. HP CoolTown project [13] is one solution that follows this decoupling. It identifies each person, place and thing with a unique *Web Presence* location and maintains relationships among these entities through the use of Web Presence Manager (WPM).

Although the Web Presence Manager provides an intrinsic example of a Web Service for aggregating context-information, it suffers from the following limitations:

The Web Presence Manager maintains information about *physical domains* only. For example, it can store a link between Bob and a conference room, but it cannot readily describe a relationship between Bob and an ongoing meeting inside that conference room. The *Me-Centric Domain Server*, on the other hand, maintains information about any *domain* category, *i.e.* information about *physical domains* as well as the *virtual domains*.

The Web Presence Manager stores information in a XML [3] format only. Although, this provides the Web Presence Manager with a commonly adopted exchangeable syntax, the annotation has no semantic meaning. The information thus does not allow deduction of additional contextual knowledge. For example, the Web Presence Manager can again store a link between Bob and a conference room. However, it is unable to automatically deduce additional relationship that Bob is inside a room. To overcome this limitation, the *Me-Centric Domain Server* is designed to use a semantically rich language.

In fact, the Web Presence Manager serves as a repository only. It does not automatically infer any contextual knowledge based on the existing information. The *Me-Centric Domain Server* on the other hand is designed to reason over its knowledge and add as many of additional implicit information as possible.

Lastly, the Web Presence Manager limits the amount and type of information that it can store. This is given by the fact that it stores XML-encoded data only. Moreover, a data that is defined by "templates" and data that the Web Presence Manager does not reason over. This limitation implies that a change to a "template" requires a change to the code implementing the Web Presence Manager. This in turn implies that subsequent versions of Web Presence Manager may be incompatible. In contrast, the *Me-Centric Domain Server* does not impose any restriction on the type and/or amount of information. Instead, it functions as a processing engine and accepts any information encoded in a semantically rich language. It then utilizes any available information for deducing all possible explicit and implicit contextual knowledge. For example, the same instance of *Me-Centric Domain Server* can infer relationships among things present in a building as well as infer that a person is vegetarian. To accomplish it, the user of *Me-Centric Domain Server* must only upload to the service an information about each entity, *i.e.* things and people, and *domain* descriptions including rules, *i.e.* building and vegetarian *domains*.

```
<!ELEMENT Employee (name)>
<!ELEMENT Client   (name)>
<!ELEMENT name     (#PCDATA)>

<Employee>Bob</Employee>
<Client>Alice</Client>
```

Figure 3.1: XML excerpt representing Bob is an employee, and Alice is a client.

```
<rdfs:Class rdf:ID='Person' />
<rdfs:Property rdf:ID='name' />
 <rdfs:domain rdf:resource='Person' />
</rdfs:Property>
<rdfs:Class rdf:ID='Employee'>
 <rdfs:subClassOf rdf:resource='Person' />
<rdfs:Class>
<rdfs:Class rdf:ID='Client'>
 <rdfs:subClassOf rdf:resource='Person' />
<rdfs:Class>

<Employee rdf:about='B007'>
  <name>Bob</name>
</Employee>
<Client rdf:about='A012'>
 <name>Alice</name>
</Client>
```

Figure 3.2: RDF(S) excerpt representing Bob is an employee, and Alice is a client.

Consequently, we have designed the *Me-Centric Domain Server* as a generic service that aggregates and also reasons over context information. The *Me-Centric Domain Server* annotates this information using a semantically rich language and tries to overcome all other limitations of the Web Presence Manager.

3.2 On Semantic Mark-Ups and Their Advantage

In the previous section, we have argued that the use of XML and other syntax-only languages is insufficient to represent contextual information. Instead, we claim that the use of a semantically rich language for annotating the information is necessary. To illustrate this requirement, let us look at Bob and Alice. Bob is an employee of a certain company and Alice is a client of the same company. Figure 3.1 defines this knowledge using XML and Figure 3.2 illustrates the same information annotated in Resource Description Framework and Schema (RDF(S)) [14, 4]. In addition and for the sake of simplicity, let us consider that we want to ask for all people and list their names. The only way to answer this query using the XML representation of information about Bob and Alice involves a human and writing of a new program. This program must be manually tailored to access the appropriate attributes according to the XML Document

Type Definition (DTD). On the other hand, querying the RDF(S) representation is quite simple. Although, the querying application may still not understand what the keywords `Employee`, `Client` and `Person` mean, it knows from the RDF(S) definitions that both `Employee` and `Client` classes are subclasses of `Person`. This implies that every instance of an `Employee` or a `Client` must be implicitly an instance of a `Person`. Accordingly, by directly querying the RDF(S) representation we immediately receive the names of Alice and Bob.

With this in mind, we have designed the *Me-Centric Domain Server* to employ a semantically rich language. For our initial implementation, we have considered RDF(S). RDF(S) is a standard defined by the World Wide Web Consortium. By choosing RDF(S), the *Me-Centric Domain Server* still provides a mean for high inter-operability with other Web Services and at the same time it can utilize the RDF(S) metadata for its reasoning.

Chapter 4

Approach

In this report, we present our work on the *Me-Centric Domain Server* ($M_{er}C_eD_eS$). The task of $M_{er}C_eD_eS$ is to act as a focal aggregation point of context information. It collects any information from sensors and others sources. It reasons over the aggregated information and detect additional contextual knowledge. Lastly, it makes the information available for querying by any service, application and/or agent that wishes to utilize it. Our approach therefore consisted of the following steps:

1. A study of existing context-aware architectures.

   In this initial step, we have examined existing approaches that have dealt with collecting, storing, and inferring context knowledge, such as the HP CoolTown Project, the Context Toolkit [7], and the Bat Teleporting [12]. As mentioned in the Chapter 3, we have concluded that $M_{er}C_eD_eS$ will require context information annotated in a semantically rich language. The use of a language at a syntax level is insufficient because it does not allow for additional reasoning. We have also concluded that $M_{er}C_eD_eS$ should use a database as its underlying storage capacity. This allows $M_{er}C_eD_eS$ to store a large amount of information while at the same time provide a quick querying functionality. Since $M_{er}C_eD_eS$ may have to store a large amount of information, the use of main memory is not sufficient. Additionally, since $M_{er}C_eD_eS$ may have to support a large number of concurrent clients the use of files

for storing the information is insufficient. Lastly, $M_{er}C_eD_eS$ also benefits from a transaction support of an underlying database because many clients may be modifying $M_{er}C_eD_eS$ internal knowledge base concurrently.

2. A study of the use of RDF(S) as the means for defining and utilizing context information.

   For our initial implementation of $M_{er}C_eD_eS$, we were deciding among the existing semantic languages, namely the RDF(S) and DAML+OIL. We have therefore examined existing use cases that use one of these languages, such as the UMBC xTalks [6], the DMOZ Open Directory [18]. We have studied how the language was used in these systems in terms of defining ontologies and how the metadata was utilized. For example, the xTalks uses the semantic language to annotate events and peoples profiles, and then uses the metadata to find events that match people's preferences. We have also studied how the systems benefited from using a semantic language instead of a syntax language only. For example, the DMOZ Open Directory uses the *subClassOf* feature to define that any movie web site is also an entertainment web site. On the other hand, using a syntax language would require that each movie is explicitly declared to be of both types. Noticing that there are not many differences between RDF(S) and DAML+OIL, we have concluded that RDF(S) is sufficient for our requirements. Having said that we first considered DAML+OIL.

3. A study of existing RDF storage architectures,

   We have initially considered the use of DAML+OIL and have evaluated two architectures for storing DAML-annotated information in a Berkeley DB [20] database; however, the possibilities were fairly limited. We have therefore turned to using RDF(S) only. For our first version we have decided to use the HP Jena Toolkit [15], which had a direct support for a relational database. However, as we explain in Chapter 7, we were faced with long time delays during information processing. We were therefore forced to revisit our design and implement a new database support, which we describe in Chapter 5.

4. Definition of *Entities*, *Domains* and their relationships.

Since we have selected RDF(S) as the semantic language of our choice, we have converted the concepts of *entities*, *domains* and relationships among them into RDF(S) ontologies. We have defined *domain* to consist of its human-readable description, a set of requirement functions used for determining memberships, and a set of policies that are applicable to member entities. We have defined *domain set* properties among the *domains*, such as *complimentOf* or *disjointWith*. We have also defined three categories of functions that are used for the membership calculation. We formally introduce these concepts in Chapter 5.

5. Definition of interfaces and actions that are to be visible to external services, applications and agents.

   Having defined the types of information and the means for inferencing over the aggregated knowledge, we then proceeded with defining a messaging interface with string messages based on the FIPA ACL message representation [11]. As we describe in Section 5.5.1, the message primarily consists of information about sender, receiver, speech-act and the content. The content is used to encode query, where a client is asking for relationship among *entities* and *domains* as well as for modifying the $M_{er}C_eD_eS$ internal knowledge base. We have also defined a set of speech-acts that correspond to the possible actions, a client can request $M_{er}C_eD_eS$ to perform. We introduce these actions in Section 5.3.2.

6. Implementation, refinement, and experiment.

   After implementing and initial version of $M_{er}C_eD_eS$, we have designed and implemented an experimental scenario situated in an office-environment. In this scenario, we are able to track locations of the company employees. According to the sensed locations and to *domain* membership rules, $M_{er}C_eD_eS$ infers a set of additional context information such as that people are talking or that they are busy. We provide a description of the scenario and testing results in Chapter 7.

Chapter 5

Design and Specification

In this chapter we describe the design and specification of the *Me-Centric Domain Server* ($M_{er}C_eD_eS$). We present the concepts that are used to aggregate, reason over, and infer contextual information. We describe the ontologies that represent this defined terminology. We define the necessary operations that should be made available for an external use by other applications and services. We describe our design choice for the $M_{er}C_eD_eS$ architecture, and define the interaction interface and messaging protocol.

5.1 Concepts

The two primary concepts used in the *Me-Centric Domain Server* for defining context knowledge are *entity* and *domain*. Here, we formally define these two terms as well as possible relationships among them.

5.1.1 Entity

We first define the notion of an *entity*, which we use to represent objects of our interest. For example, a person, a baseball team and night are all valid *entities*.

> **Definition** Entity is any uniquely identifiable thing or an abstract term for which we may require a knowledge of its relevant context.

5.1.2 Domain

We next define the notion of a *domain*. We use *domain* as an abstraction for a specific part of the currently examined environment. Moreover, each *domain*'s coverage area is bounded by in terms of property requirements and each *domain* is associated with applicable policies.

> **Definition** Domain is a uniquely identifiable abstract term that defines a set of necessary requirements that are used to define an entity membership and that defines a set of policies which are applicable to member entities.

According to this definition, we can define a context of each entity as a collection of domains. For example, we can use a *food domain*, a *vegetarian domain* and a *restaurant domain* to define the relevant context when a person enters a restaurant and a waiter is trying to suggest the most appropriate meal.

As mentioned in Chapter 1 we can further differentiate among four domain categories [5]:

> **Definition** Physical Domain represents a domain that is directly linked to the physical world with well-defined borders.

An example of a *physical domain* is any building and even a particular cell of a cellular network.

> **Definition** Knowledge Domain represents a domain whose members are explicitly defined and a membership is determined through a computation or a file system access only.

An example of a *knowledge domain* is a database of company employees, a yellow page service, the fact that a person is vegetarian, or the time of the day. However, building and meeting are not example of a *knowledge domain* because their membership depends on physical sensing and combining multiple domains, respectively.

> **Definition** Conceptual Domain represents a domain where an entity membership can neither be directly sensed nor computed but is instead determined by combining information obtained from other domains.

An example of a *conceptual domain* is a going-home-domain which requires the knowledge of not only that a person is in her car but also the direction, time of the day, etc.

> **Definition** Collaborative Domain is a domain that captures membership information for a group as opposed of membership information for an entity at a time.

An example of a *collaborative domain* is a meeting, which depends on detecting activity of more than one entity, in this case more than one person.

5.1.3 Membership Relations and Rules

Having defined the notion of *domain* and *entity*, we can now define the notion of a *membership* among them.

> **Definition** Given entity e, domain D and its corresponding boolean requirement function R_D, e is member of a D, denoted by $e \in D$, if e satisfies the necessary requirements of D, i.e. $R_D(e) = true \Rightarrow e \in D$.

We therefore require a boolean function which given an entity and a domain maps into true whenever an entity should be a member of the particular domain, and otherwise the function must map into false.

> **Definition** Function F is a requirement function for domain D, when for every entity $e : F(e) = true \iff e \in D$.

For our purposes we differentiate among three categories of requirement functions, a predicate function, an AND function and an OR function.

> **Definition** Predicate function P is a requirement function for domain D, when $\forall e : P(e) = true \iff P(e)$ can be satisfied by inferring over the aggregated contextual information.

> **Definition** AND function $A(X, Y)$ is a requirement function for domain D, when $\forall e : P(X, Y) = true \iff X = true \wedge Y = true$ where X and Y are requirement functions for domain D.

> **Definition** OR function $A(X, Y)$ is a requirement function for domain D, when $\forall e : P(X, Y) = true \iff X = true \vee Y = true$ where X and Y are requirement functions for domain D.

As we illustrate in the later chapters, these three types of functions allows us to construct almost any requirement for calculating and deducing domain memberships.

5.1.4 Domain Set Theory

In addition to the above functions, we also define a set of simple properties for describing relationships among *domains*. These allow us to implicitly define additional requirements without the burden of defining each rule over and over again for every *domain*. For example, by defining a domain *inCubicle* as being a subset of a domain *in-office*, we can directly infer that all people currently present *inCubicle* are also *in-office*. We define the following set of domain set properties:

> **Definition** Domain D equals to D_2, denoted $D \equiv D_2$, $\iff (\forall e : e \in D \iff e \in D_2)$.

> **Definition** Domain D disjoint with D_2, denoted $D \parallel D_2$, $\iff (\forall e : e \in D \Rightarrow e \notin D_2 \wedge \forall f : f \in D_2 \Rightarrow f \notin D)$.

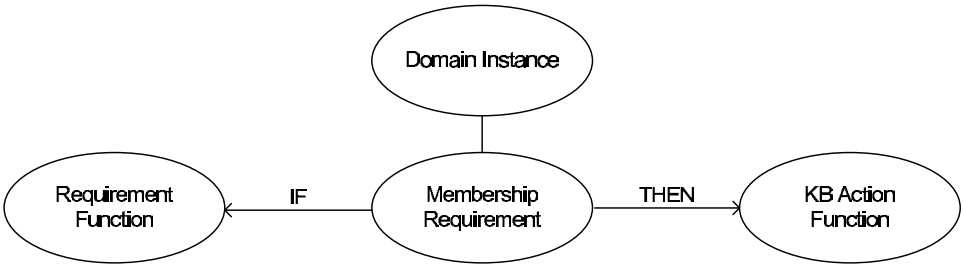


Figure 5.1: RDF(S) graph tree representing the *KBIfThenCondition* class.

Definition Domain D is complement of D_2, denoted $D \mid D_2$, $\iff (\forall e : e \in D \iff e \notin D_2)$.

Definition Domain D is a subset of D_2, denoted $D \subset D_2$, $\iff (\forall e : e \in D \Rightarrow e \in D_2)$.

Definition Domain D is an intersection of $D_2, D_3, ...$, denoted $D \bigwedge \{D_2, D_3, ...\}$, $\iff (\forall e : e \in D \iff e \in D_2 \wedge e \in D_3 \wedge ...)$.

Definition Domain D is an union of $D_2, D_3, ...$, denoted $D \bigvee \{D_2, D_3, ...\}$, $\iff (\forall e : e \in D \iff e \in D_2 \vee e \in D_3 \vee ...)$.

5.2 Me-Centric Ontology

The Me-Centric Ontology is a formal explicit description of the above defined concepts in terms of classes and properties according to the RDF(S) standards. The ontology plays a very important role in the $M_{er}C_eD_eS$ implementation as the domain rules and properties are reasoned over according to the ontological definition. The ontology does not only provide the means for exchanging information and requesting actions among services, which is equivalent to an XML-based approach. In addition, it provides the ability to automatically deduce additional domain memberships. In other words, it allows us to infer contextual information that would be otherwise inaccessible. We have discussed the reasons in Chapter 3.

The ontology describes the necessary vocabulary for defining *entities*, *domains*, the requirement functions and the domain set properties. With the inherent features of RDF(S), each entity, domain, and function are globally identifiable. This allows $M_{er}C_eD_eS$ to correctly combine sensed and inferred information from multiple sources. As we are primarily interested in the *domain* information, below we omit the description of the *entity* part of the ontology.

```
<m:Domain rdf:about="&d;atwork">
  <m:dmembershipRequirement rdf:resource="&d;dmr" />
</m:Domain>

<m:KBIfThenCondition rdf:about="&d;dmr">
 <m:if rdf:resource="&d;ifpart" />
 <m:then rdf:resource="&d;thenpart" />
</m:KBIfThenCondition>

<m:ANDFunction rdf:about="&d;ifpart">
 <m:first>
  <m:VerifyFunction rdf:about="&d;V1">
   <m:verify>
    <m:Statement>
     <m:S rdf:resource="&d;variable" />
     <m:P rdf:resource="&m;isMemberOf" />
     <m:O rdf:resource="&office;#InCubicle" />
    </m:Statement>
   </m:verify>
  </m:VerifyFunction>
 </m:first>
 <m:rest>
  <m:ORFunction rdf:about="&d;O1">
   <m:first>
    <m:VerifyFunction rdf:about="&d;V2">
     <m:verify>
      <m:Statement>
       <m:S rdf:resource="&d;variable" />
       <m:P rdf:resource="&m;isMemberOf" />
       <m:O rdf:resource="&time;#Morning" />
      </m:Statement>
     </m:verify>
    </m:VerifyFunction>
   </m:first>
   <m:rest>
    <m:VerifyFunction rdf:about="&d;V3">
     <m:verify>
      <m:Statement>
       <m:S rdf:resource="&d;variable" />
       <m:P rdf:resource="&m;isMemberOf" />
       <m:O rdf:resource="&time;#Afternoon" />
      </m:Statement>
     </m:verify>
    </m:VerifyFunction>
   </m:rest>
  </m:ORFunction>
 </m:rest>
</m:ANDFunction>

<m:KBAssertAction rdf:about="&d;thenpart">
 <m:add>
  <m:Statement>
   <m:S rdf:resource="&d;variable" />
   <m:P rdf:resource="&m;isMemberOf" />
   <m:O rdf:resource="&d;atwork" />
  </m:Statement>
 </m:add>
</m:KBAssertAction>

<m:Variable rdf:about="&d;variable" />
```

Figure 5.2: RDF(S) graph tree representing the excerpt in Figure 5.3.

Each *domain* is identified by a global Unique Resource Identifier (URI) according to the W3C standards. Each domain contains a static list of entities that are always members of this particular domain, and whose membership does not have to be re-calculated. Additionally, each domain contains a list of *KBIfThenCondition* resources. Each *KBIfThenCondition* consists of one *VFunction* that represents one of the Predicate, AND or OR functions, and it also includes a list of *KBActions*. The *KBActions* should be executed anytime the associated *VFunction* is true. We illustrate this in Figure 5.1.

The defined information in each domain instance is processed by $M_{er}C_eD_eS$ and all requirements may be periodically evaluated. In addition, every received sensed information or action may optionally cause the system to also evaluate all requirements. Accordingly, $M_{er}C_eD_eS$ always adapts its knowledge base to the latest appropriate situation.

5.2.1 Ontology Example for Domain Instance

To better illustrate how the ontology is used to define a domain instance, let us consider the RDF(S) excerpt in Figure 5.3 and its graphical representation in Figure 5.2. The excerpt defines an *atwork* domain that

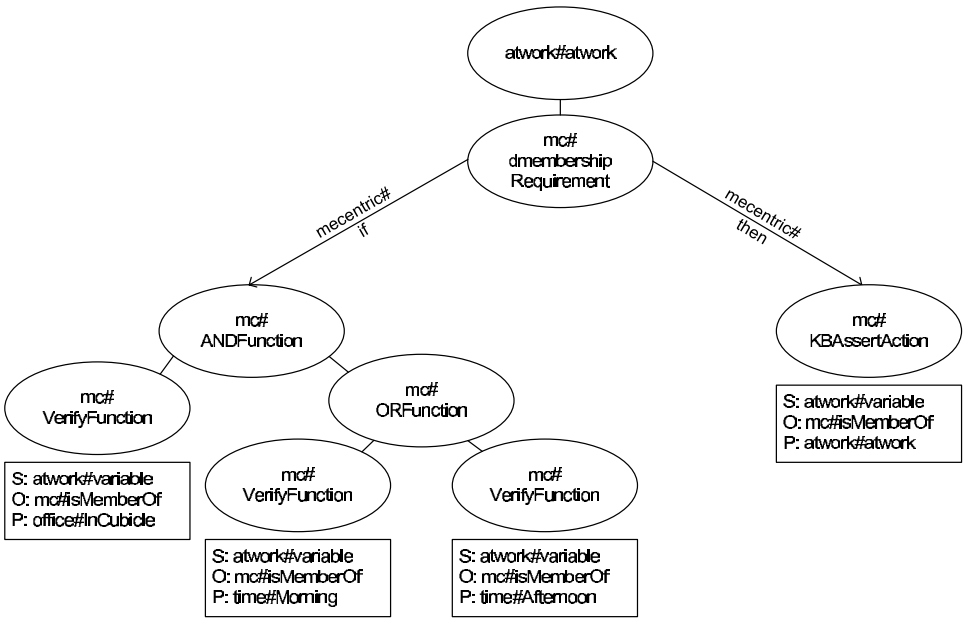


Figure 5.3: RDF(S) excerpt of the *atwork* domain.

has a membership requirement defined in terms of one AND function, one OR function and three predicate functions. The combined requirement function implies that an entity can be member of this domain only when it is a member of *morning* or *afternoon* domain and at the same time the entity must be a member of *inCubicle* domain.

5.3 Me-Centric Domain Server Architecture

Having specified the necessary concepts as well as defined the necessary actions that the $M_{er}C_eD_eS$ service should perform in order to aggregate and interpret contextual information, let us now describe the architecture design of $M_{er}C_eD_eS$.

Figure 5.4 represents an overview of our architecture design. The $M_{er}C_eD_eS$ service consists of two main components in addition to the RDF(S) ontologies. First, the $M_{er}C_eD_eS$ service includes an RDF(S) database, which is used to store all information about each *entity*, *domain*, and their associated relationships. Next, the $M_{er}C_eD_eS$ service includes a pool of processes, *i.e.* threads, which are responsible for interacting

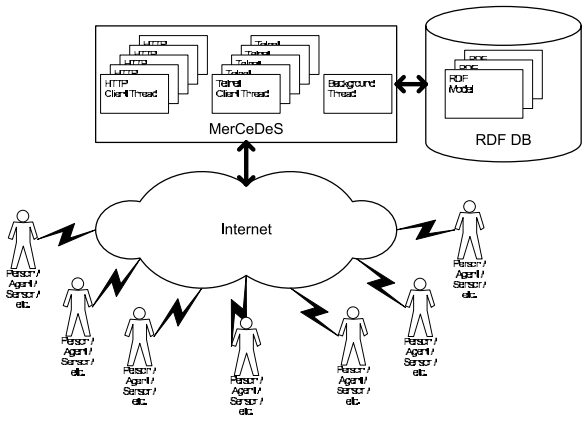


Figure 5.4: Me-Centric Domain Server Architecture

with other services, applications, and agents. These threads are responsible for accepting queries, updating the $M_{er}C_eD_eS$ internal knowledge base, and for informing clients about the appropriate results. In addition, there exist a specialized set of processes, which is optionally responsible for a periodic inferencing over the stored knowledge base. Accordingly, the knowledge base reasoning may be periodic and/or triggered by adding/removing information.

5.3.1 Process Flow

As mentioned above we can differentiate among two types of processes. One category, *i.e. client threads*, is responsible for interacting with other services, while the second category, *i.e. background threads*, is responsible for the internal modification of the global knowledge base. We therefore differentiate among two types of process flows as illustrated in Figure 5.5.

The *client threads* continuously execute four distinct steps. First, they are waiting for other services to submit an action request. Once a message is received, the appropriate *client thread* parses the message and decides what actions should be taken. It then converts the action into a query and executes the query on the internal knowledge base. Upon either success or failure, the thread constructs a response message and sends it back to the requester. In this approach, the *client threads* are not required to keep a communication session. They are thus very efficient for both a direct telnet-like communication and as a Web Service.

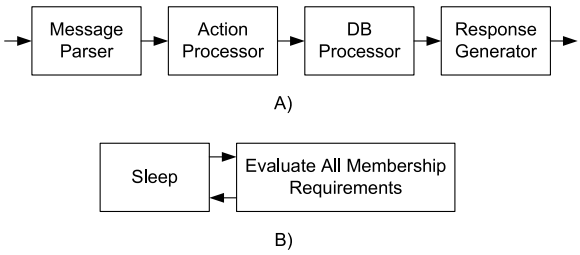


Figure 5.5: Process Flow

The *background threads* are responsible for a periodic reasoning over all rules and domain set properties in order to detect any additional membership that was not specified explicitly. By decoupling this expensive execution from the *client threads* the $M_{er}C_eD_eS$ service is able to satisfy more concurrent users more frequently, although the responses may be relatively stale at times. At the same time the client can still explicitly ask $M_{er}C_eD_eS$ to re-infer over the knowledge base.

The tradeoff of this approach is that the stored information does not always reflect the current situation. In fact this approach imposes a short update delay that is equivalent to the time period the background threads are inactive. On the other hand, this approach does not require reasoning over the entire knowledge base anytime an information is added and/or removed.

5.3.2 Me-Centric Domain Server Actions

We now define the actions that the $M_{er}C_eD_eS$ service makes publicly available to be employed by external services, which are list in Figure 5.6.

Domain Instance Maintenance

The first set of actions allows external services to manage the static information about each *domain* instance. This allows each service to add new *domain* instance by uploading a new RDF document. This also allows a service to update an existing *domain* instance by adding a new list of static members or by modifying the requirement functions. Naturally, it also enables a service to remove an existing *domain* instance from the $M_{er}C_eD_eS$ internal knowledge base.

```
domain
      add-domain
      update-domain
      remove-domain

entity
      add-entity
      update-entity
      remove-entity

fact
      add-membership-information
      remove-membership-information

inference
      infer-domain-members
      remove-domain-members
      infer-entity-memberships
      remove-entity-memberships

query
      ask-if-member

response
      response-ok
      response-error
```

Figure 5.6: Supported Me-Centric Domain Server Actions

Entity Instance Maintenance

Similarly to the operations on *domain* instances, the $M_{er}C_eD_eS$ service permits any external service to add, modify and remove any *entity* instance.

Asserting and Sensing

Until now we have been talking about actions that modify the static part of the internal knowledge base. In addition to those, the $M_{er}C_eD_eS$ service defines two operations, namely `add-membership-information` and `remove-membership-information`, which allows a sensor to update the internal knowledge base by informing the $M_{er}C_eD_eS$ service about any changes it has recently sensed.

Inferencing

In the case, when an external service requires an instant recalculation of dynamic, *i.e.* inferred, membership information about some *entity* or *domain*, the $M_{er}C_eD_eS$ service also allows such external service to specify the URL of the desired *entity*/*domain*. Upon receipt of such request, the $M_{er}C_eD_eS$ *client thread* reexamines the knowledge base in the same manner as the *background thread*, and it returns a list of appropriate matches.

Querying

The last set of actions allows the external services to query the internal knowledge base. A service can ask for all members of a particular *domain*, *e.g.*, "Who is in domain at-work?" A service can also ask for all memberships of a particular *entity*, *e.g.*, "Where is Bob?" Next, a service can ask whether a particular *entity* is member of some *domain*, *e.g.*, "Is Bob at at-work?" Lastly, a service can also ask for all known memberships and will receive every match from the internal knowledge base.

5.4 DB RDF Storage

As mentioned above, we have chosen to employ a relational database for the internal knowledge base. In our initial implementation we were solely dependent on the use of HP Jena 1.4.0 Toolkit, which already supported a database storage on its own. However, after a set of experimental testings and for reasons which we describe in Chapter 7, we have decided to revisit our design and create a new database support. Additionally, we have defined our own breed of an RDF query language and designed a simple algorithm for converting RDF queries into their corresponding SQL counterparts.

5.4.1 Table Definition

Inside the database, we use multiple RDF models. This concept was previously introduced in the HP Jena Toolkit to separate among a number of RDF documents that can be stored in the same table. For example, this way we guarantee that information about Bob is not mixed with information about Alice, which could otherwise lead into unexpected system behavior. The $M_{er}C_eD_eS$ database therefore consist of two tables whose structure is depicted in Figure 5.7. The `rdf_models` table is used for storing a list of all RDF models that have been so far defined during the runtime of $M_{er}C_eD_eS$. The `rdf_modelspo` table is used for storing each RDF `subject-predicate-object` tuple associated with an appropriate RDF model. Additional information for each RDF tuple includes three bits, namely *isL*, *isN*, and *isV*. The *isL* bit should be set to true whenever the *object* of the tuple is a literal as opposed to an RDF Resource. The *isN* bit

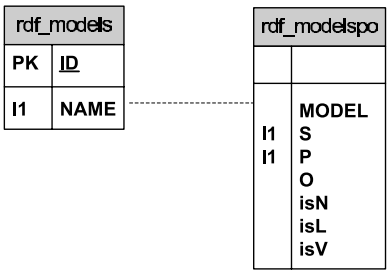


Figure 5.7: Me-Centric Domain Server Database Structure

should be set to true whenever the *object* of the tuple is numeric. Finally, the isV bit should be set to true whenever the *object* represents an variable, *i.e.* an instance of a the class *Variable*. We have chosen this naive approach for the table definition in order to increase the SQL query processing to its maximum. On the other hand, the tradeoff of our approach is that it is not storage efficient as every tuple includes the full name-space and local name for its subject, its predicate and its object.

5.4.2 RDF Models

In previous section we have mentioned the notion of RDF models. The HP Jena Toolkit first introduced this concept which enables an application to separate RDF statements associated with one document or some object from others. We have applied a similar principle to define a distinct RDF model for every *domain* and *entity* instance. This allows us to quickly add, update and/or remove only the appropriate set of RDF statements without modifying any other stored information. In addition, we define an RDF model for storing all sensed information, and another RDF model for storing the deduced information. In this way, we can easily differentiate among static information, *i.e.* information stored in the *domain* and *entity* instance models, and dynamic information, *i.e.* the sensed or deduced knowledge. This distinction is necessary as the $M_{er}C_eD_eS$ service does not only add knowledge but may also modify or even retract information.

5.4.3 Converting RDF Rules to SQL Queries

To be able to quickly and efficiently infer any additional *domain* membership, the $M_{er}C_eD_eS$ service must be able to parse and process every `KBIfThenCondition` object present in the database. As the knowledge base is represented by a relational database, we have focused on an algorithm for a quick compilation of the `PredicateFunction`, `ANDFunction` and `ORFunction` objects into fast SQL queries. Although the algorithm is simple, we show in Chapter 7 that combined with our database structure it is very efficient. The algorithm consist of the following steps:

1. If the evaluation function is already cached, proceed to step 8.
2. Parse the RDF model.
3. Create a tree representation of the evaluation function.
4. Flatten the tree by traversing it in a depth-first search manner, while
   (a) For every `PredicateFunction` create a new list with this function as the only element;
   (b) For every `ANDFunction` create a new list by adding each combination by one element from its left child to one element of its right child; and
   (c) For every `ORFunction` append the list from its right child to the list of its left child.
5. Every row of the resulting list contains only `PredicateFunction` objects that have to be all true for the row to evaluate as true, and no row depends on another one.
6. Therefore, for every row in the list, construct an SQL joint query that returns tuples of variable assignments for which the row is true according to the database.
7. Cache all resulting queries.
8. Execute each cached SQL query associated with this evaluation function, and assert/retract RDF statements according to the `KBAction` part of the appropriate `KBIfThenCondition`.

5.5 Interaction

The last aspect of our architecture consists of designing appropriate interaction protocols with external services and a message format. To allow almost any application to interact with the $M_{er}C_eD_eS$ service, we have chosen to support a telnet-like interface as well as to support HTTP `GET` and `POST` methods. The telnet-like interface allows any application to connect to $M_{er}C_eD_eS$ on a well-known port and exchange message through the open socket connection. While the HTTP interface allows applications to interact using the

```
<message> =   ( :speech-act "<speechact>"
                :sender "<sender>"
                :receiver "<receiver>"
                :date "<date>"
                :language "<language>"
                :reply-with "<reply-with>"
                :in-reply-to "<reply-with>"
                :content <item> | <list> )

<speechact> =   add-domain | add-entity | add-membership-information | ask-if-member |
                infer-domain-members | infer-entity-memberships | remove-domain |
                remove-domain-members | remove-entity | remove-entity-memberships |
                remove-membership-information | update-domain | update-entity |
                response | response-ok | response-error

<sender> =      {SENDER_ID}

<receiver> =    {RECEIVER_ID}

<date> =        {DATE}

<language> =    mecentric

<reply-with> =  {REPLY_MSG_ID}

<in-reply-to> = {REPLY_MSG_ID}

<item> =        (:<attribute> "<value>" :<attribute> "value" ...)

<list> =        (:list (<item>,<item>,...,<item>))

<attribute> =   {ATTRIBUTE_NAME}

<value> =       {ATTRIBUTE_VALUE}
```

Figure 5.8: Me-Centric Domain Server Message Format

standard HTTP protocol. Additionally, we have based our message definition on the FIPA ACL message representation specifications [11] to enable for a future support of traditional agent-based communication.

5.5.1 Message Format

In designing the message format, we have followed the FIPA ACL message representation specifications. At the same time the goal of the initial version was to make it simple. Additionally, since the $M_{er}C_eD_eS$ by default supports primarily an IP telnet-like and HTTP protocols, some fields were unnecessary. We have therefore selected only the fields that are directly relevant to the $M_{er}C_eD_eS$ service. These fields include a

header information about the requested speech act, identification of both sender and receiver, information allowing for message ordering, and the content information. Based on the different types of actions, *i.e.* speech acts, we can generalize the content of the message to include a list of URLs associated with specific attribute names and/or a name-value attribute for representing an RDF document. We illustrate the message format in Figure 5.8. Note that the major difference between the FIPA ACL format and our design is the definition of speech act. While FIPA ACL requires that the speech act type is the first word after the opening parentheses, in our case, the speech act is treated as another name-value pair, with name set to `speech-act`.

5.5.2 Interaction Protocol

As mentioned above, the $M_{er}C_eD_eS$ service currently supports two types of interaction. One approach is based on HTTP protocol. In contrast, the other approach is based on a simple telnet messaging with opening a socket connection among the $M_{er}C_eD_eS$ service and the requesting application. The interaction protocol is very naive as it operates in a request-response mode.

To allow even the HTTP approach to send more than one request per connection we have defined two additional communication commands, namely: `EOL_COMMAND` and `CLOSE_COMMAND`. The first command is used to separate multiple requests per connection, while the latter command is used to inform the other party that one direction of the connection was closed. Therefore, in the case of the HTTP-based interaction, the client may request multiple actions by submitting (message 1) `EOL_COMMAND` (message 2) `EOL_COMMAND` ... `EOL_COMMAND CLOSE_COMMAND` using either the POST or GET standards, and it will receive all appropriate responses in body of the incoming HTML document.

Chapter 6

Implementation

In this chapter we present the current implementation of the *Me-Centric Domain Server* ($M_{er}C_eD_eS$). We have developed and tested $M_{er}C_eD_eS$ on an Intel Pentium IV 1.8GHz machine with 256MB RAM memory running Microsoft Windows 2000. The $M_{er}C_eD_eS$ source consists of several thousand lines of Java code. We have used the Sun J2SDK version `1.4.0` for Windows for both compiling and executing $M_{er}C_eD_eS$. $M_{er}C_eD_eS$ utilizes MySQL relational database [1] for its underlying storage capacity. We have used the MySQL server version `3.23.51-max-nt` for Windows and set the $M_{er}C_eD_eS$ tables to *MyISAM* type. We have used HP Jena Toolkit version `1.4.0` for parsing RDF documents into RDF `subject-predicate-object` statements. We have also used Apache Tomcat HTTP [2] server version `3.3.1` for Windows for the $M_{er}C_eD_eS$ HTTP interface. Figure 6.1 summarizes where these external toolkits were used. The code accompanied by documentation is available at `https://source/pCache/sdatamecentric.htm`.

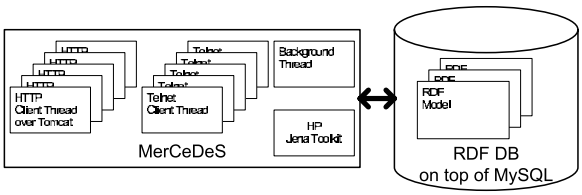


Figure 6.1: Use of external toolkits within the Me-Centric Domain Server

6.1 Me-Centric Domain Server Code Library

The $M_{er}C_eD_eS$ main code is logically organized into three sections, *i.e. packages*. One *package* implements necessary methods for message processing based on the design from Section 5.5.1. This includes a creation of new messages, parsing existing messages, and also access and modification of their contents. Next *package* is responsible for client interaction and performing actions on client's behalf that were specified in Section 5.3.2. The third *package* is responsible for modifying $M_{er}C_eD_eS$ internal knowledge base and for querying it.

6.1.1 `com.hp.hpl.mecentric.message` Package

This *package* implements code for performing message operations According to the specification from Section 5.5.1, each message consists of *name-value* attributes. Additionally, the `content` attribute also consists of a number of *name-value* pairs.

The `MessageConstants` class file specifies all legal attribute names that can exist inside the $M_eCentric$ message. This file additionally specifies all legal speech-act types, such as `add-domain` and/or `ask-if-member`, and other message constants.

The main class file of this *package* is `Message`. It provides methods for parsing an existing message as well as for creating a new message. It can create a new blank message or a *reply* message by reversing the sender and receiver information, and by setting the `in-reply-to` attribute accordingly to the original message. It also provides all *SET* and *GET* methods to modify message attributes.

The *package* thus serves as a generic interface to message processing, and can be used independently from the $M_eCentric$ code. This allows other applications to reuse this code in order to be able to interact with $M_eCentric$.

6.1.2 `com.hp.hpl.mecentric.domainserver` Package

This *package* implements code for interacting with clients and for performing actions on their behalf. It specifies the telnet-based and HTTP interface for accessing $M_{er}C_eD_eS$ knowledge base.

The main class file for telnet interaction is `DomainServer`. Upon startup it creates a pool of *client* threads according to the design specification in Chapter 5. It listens on a well-known port, `8881` by default, for connection requests and assigns clients to its threads. Each thread then processes the message request, performs an action, and sends results to the client.

The `HTTPDomainServer` is an optional wrapper class around `DomainServer`. It is used together with Tomcat server to provide HTTP interaction access. This class is extended by `HTTPDomainServerDemonstrator` to provide a em pretty-print functionality. We use this extension for our demonstrating experiments in Chapter 7.

6.1.3 `com.hp.hpl.mecentric.domainserver.rdb` Package

The third *package* implements code for modifying and querying the $M_{er}C_eD_eS$ internal knowledge base. It implements internal database access methods and is responsible for maintaining consistency among stored assertions.

The main class of this package is the `SQLModel`. It provides generic methods for creating and modyfing *domain* and *entity* instances. It also provides methods for *querying* the context information. Lastly, it provides an interface for reasoning over the information and deducing additional knowledge.

For reasoning, the `SQLModel` class depends primarily on the `RDQLFunction` and `KBAction` classes. `RDQLFunction` retrieves and compiles RDF *requirement functions* into SQL queries. The class also executes the queries and caches matches. Based on these matches, `KBAction` then asserts and/or removes additional implicit knowledge into the database.

6.2 Demonstrator Code Library

We have developed additional code for demonstration and experiment purposes. This library includes source code mainly for developing clients. Although, it also includes the `HTTPDomainServerDemonstrator` class. As we have mentioned above, this class is used for a *pretty-print* functionality. It is designed for

demonstration only to allow humans to better comprehend the content of each request and response message. In fact, during our HTTP-based demonstration one can switch between using `HTTPDomainServer` and `HTTPDomainServerDemonstrator` at any time.

6.2.1 `com.hp.hpl.mecentric.domainclient` Package

This *package* includes an optional code that is not used by $M_{er}C_eD_eS$ at all. It only specifies a set of useful methods for telnet-based interaction with $M_{er}C_eD_eS$. It allows an application using this *package* to establish and maintain a telnet connection. Additionally, it provides methods for sending and receiving messages from the $M_{er}C_eD_eS$ service.

6.2.2 Web Client Package

This *package* consists of set of optional Java Server Page (JSP) files and images that we use for our $M_{er}C_eD_eS$ demonstrator of office environment. These files are located in the `web` directory of $M_eCentric$ source code. To use them, they must be uploaded to Tomcat HTTP server location.

The `/mc/` package provides a simple HTML interface for all $M_{er}C_eD_eS$ implemented actions. It provides form constructs for each message type. It employs JavaScript to convert the form into a legal *Me-Centric* message, and sends it to $M_{er}C_eD_eS$ HTTP interface.

The `/mc/office/` package also provides a simple HTML interface. In this case it is customized to reflect the office demonstration environment. It again uses JavaScript to convert office form information into a legal *Me-Centric* message, and sends it to $M_{er}C_eD_eS$ HTTP interface.

This *web client package* therefore does not modify the $M_eCentric$ server side. It only customizes the look and feel of the client side and uses a default $M_eCentric$ implementation.

6.3 Compiling and Running Me-Centric Domain Server

As mentioned at the beginning of this Chapter, the code accompanied by its documentation is available at `https://source/pCache/sdatamecentric.htm`. The source code is organized into six folders. Two folders, *i.e.* `docs` and `javadoc`, are used for documentation and Java documentation, respectively. The `web` folder includes the *Web Client* package described above. The `src` folder contains the source `*.java` files. The `classes` folder is used to contain compiled `*.class` files, and the `jar` folder is used to contain a compiled $M_{er}C_eD_eS$ package.

6.3.1 Compiling Me-Centric Domain Server

To compile the entire $M_{er}C_eD_eS$ including its HTTP interface, the HP Jena Toolkit, Apache Tomcat, and a MySQL driver must be first installed. Additionally, the MySQL server should be installed as well.

The source code includes a `Makefile` which compiles the entire code. It requires the `$CLASSPATH` to include `<TOMCAT>/lib/common/servlet.jar` file, all `<JENA>/lib/*.jar` files, and a MySQL driver such as `mm.mysql-2.0.14.jar`.

Once the `$CLASSPATH` is set, `make classes` and `make jar` creates `*.class` and `jar` files, respectively.

6.3.2 Including Me-Centric Domain Server in Tomcat

To run $M_{er}C_eD_eS$ as an HTTP servlet, the `jar` file must be compiled according to the above instructions. The resulting `jar` file together with the dependent `jar` files from previous section should be included in `web/mc/WEB-INF/lib` folder. Next, the entire `web/mc` folder should be copied to `<TOMCAT>/webapps/` location. The last step then involves modifying `<TOMCAT>/bin/tomcat.bat` by adding comments to include `/mc/` during Tomcat startup. We describe precise steps in the `README` file accompanying the source code.

6.3.3 Preparing MySQL database

$M_{er}C_eD_eS$ relies on a MySQL database for its underlying storage capacity. Thus, a MySQL server must be running and include a database with table definitions according to specification in Section 5.4.1. As part of the $M_{er}C_eD_eS$ source code, the `docs/tables.sql` file includes a dump of the used definitions.

6.3.4 Running Me-Centric Domain Server

To run $M_{er}C_eD_eS$, the code must be compiled and optionally included as a part of the Tomcat HTTP server. Additionally, the MySQL database must be already running.

To run $M_{er}C_eD_eS$ with a telnet-based interface, simply start `make run` or `make runjar`. The `Makefile` can be further modified to reflect appropriate values of start-up parameters. These parameters include number of initial threads, maximum number of threads, listening port, and information for connecting to the MySQL database.

To run $M_{er}C_eD_eS$ with a HTTP-based interface, simply start the Tomcat HTTP server. To modify start-up parameters, change the values in the `<TOMCAT>/webapps/mc/WEB-INF/web.xml` file. This file defines the same parameters as the `Makefile`; however, it differentiates among settings for `/mc/ds` and `/mc/ds_demonstrator`. Again, these parameters include number of initial threads, maximum number of threads, listening port, and information for connecting to the MySQL database.

6.3.5 Evaluating Me-Centric Domain Server

To test the telnet-based $M_{er}C_eD_eS$, simply run `make runclient`, which tries to connect to the default $M_{er}C_eD_eS$ location, *i.e.* `localhost:8881`. Upon connection, it requests an action and awaits response.

To test the HTTP-based $M_{er}C_eD_eS$, simply run `make runhttpclient`, which tries to use `wget` program to connect to $M_{er}C_eD_eS$ at `http://localhost:8080/mc/ds`. It again requests an action and awaits response. You can also use a web browser and connect to `http://localhost:8080/mc/` to start a simple web client.

Optionally, you can connect to `http://localhost:8080/mc/office/` if the appropriate *domain* instances are already present inside the $M_{er}C_eD_eS$ knowledge base. Otherwise, this requires uploading of all `docs/demonstrator/RDF/*.rdf` files. This can be done by using `add-domain` and/or `add-entity` speech-acts through the `http://localhost:8080/mc/` interface.

Chapter 7

Demonstration and Experiments

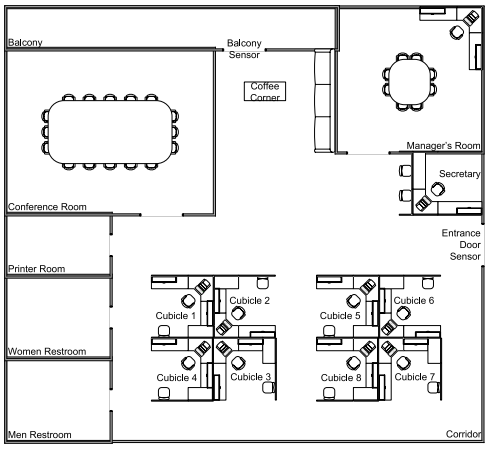


Figure 7.1: Floor plan used for the demonstrating example.

We have created a demonstration scenario situated inside an office building. We have used this scenario to illustrate the capabilities of the *Me-Centric Domain Server*. We have also used this scenario for experimenting with maintenance of the internal knowledge base. Additionally, we have extended this scenario by varying its parameters and measuring the inference speed.

The scenario describes an office building. As illustrated in Figure 7.1, the floor plan of the office covers a rectangular area of 23x21 metric units. It includes a manager room, a conference room, a printer room, restrooms, and eight cubicles. Additionally, the office includes a coffee corner, a secretary desk, and an

external balcony.

Our scenario then also includes a group of people that can randomly move within and/or outside the office. We define one manager, one secretary, and vary from 0 to 1000 additional employees.

For measuring explicit information about all office employees, *i.e.* their *physical domain* memberships, we assume a presence of two door beacons. One beacon is located at the balcony door and can correctly determine who enters/steps out the office floor through the balcony door. The other beacon is attached to the office main entrance door. It can also detect a person leaving/entering the office and is again able to identify her/him. Additionally, we are assuming a presence of a set of sensors that can triangulate a location of any person while present in the office. We simplify the triangulation to have a precision according to the office grid only. This way the sensors detect x coordinates from 1 to 23 and y coordinates from 1 to 21.

Additionally, we define a set of *domains* and their corresponding rules that use the location of people to detect their other domain memberships. We define 14 *domain* instances associated with 22 *membership requirement* rules. These domains represent each room and/or cubicles. These *domains* also include instances of virtual *domains*, such as, *at-meeting*, *meeting-with-manager*, or *talking*.

Our scenario, therefore, consists of an office associated with 14 unique *domains* and up to 1002 *entity* instances. We use three types of explicit information sources, and use the sensed/asserted information to deduce *domain* memberships for all *entities*.

7.1 Experiment 1: Knowledge Base Maintenance Performance

For the first set of measurements, we have chosen to evaluate the time it requires to update the *Me-Centric Domain Server* internal knowledge base. We have used the `floorplan.rdf` document as the base, and measured time it takes for the *Me-Centric Domain Server* to upload 888 RDF statements that are equivalent to this file. We have not measured the time for removing information because that usually requires only one SQL call. Next, we have used files with size of 2, 4, 8 and 16 times larger than the `floorplan.rdf`. We have assured that the corresponding RDF statements were increasing in size accordingly.

Number of RDF statements	1 x 888	1 x 888 (*)	2 x 888	4 x 888	8 x 888	16 x 888
HP Jena Parsing Time (ms)	207	207	398	739	1364	2652
Time modifying database (ms)	775	100 000	1562	3101	6258	12408

(*) Using HP Jena database support

Table 7.1: Measured performance for adding RDF statements.

We show the results of adding parsing and adding RDF statements in Table 7.1. For parsing the RDF documents into their corresponding RDF statements, we have used the HP Jena Toolkit. From the measured results we see that Jena can quickly parse large files and the speed remains $O(n)$ even for the largest tested file. However, when using Jena for implementing the underlying database, adding RDF statements had a relatively slow performance. It took 100 seconds to add 888 RDF statements. We have therefore chosen a different approach, which is not as storage efficient as Jena is; however, this other approach allows us to add triples quite fast. This is due to the fact that for our implementation we have chosen to store all information about one RDF statement in one table row. We thus store the name-spaces, Resource names, and literals all in one table row. This way, we see that our implementation of the knowledge base also delivers an $O(n)$ performance since it requires at most one SQL call per statement. On the other hand, the Jena implementation may require up to 28 SQL calls per statement. This is because it needs to check, and optionally add and check again, for the name-space and local-name of subject, predicate and object before the statement can be added.

7.2 Experiment 2: Inferencing Performance

For our next experiment, we have focused on measuring the speed of inferencing all implicit information by evaluating all present *domain membership* rules. We use a forward-chaining principle for our inferencing approach. In the current implementation, the engine sorts and evaluates all rules, which in turn may require

Number of Employees	M + S	M + S + 1	M + S + 10	M + S + 100	M + S + 1000
Inference time (ms)	3141	3539	7628	51641	731586
Inference loops	2.15	2.3	3.25	4	3

Table 7.2: Internal inference performance.

asserting and/or retracting of some context information, *i.e.* RDF statements. The engine then re-evaluates all rules again, and loops until no new knowledge can be added or removed.

We have varied number of employees from 0 to 1000 excluding manager and secretary who were always present, at least for the reasoning purposes. We show the results, in table 7.2.

Manager and Secretary (MS) only Time to deduce information (including SQL overhead): 3141ms Average reasoning loops: 2.15 MS + 1: 3539ms, 2.3 MS + 10: 7628ms, 3.25 MS + 100: 51641ms, 4 MS + 1000: 731586ms, 3

Although, the engine needed almost an equivalent average number of loops to assert all knowledge for 1 employee and for 1000 employees, the required time differs drastically. Although it takes only 3.5 seconds to assert all the knowledge for 1 employee, the inference takes over 731 seconds for 1000 employees. This in fact is due to the same problem why we have implemented our own database support for adding RDF statements. The explanation of this substantial difference lies in the fact that inference engine must check for the presence of a statement before it can add it or removed it. It uses this checking to determine whether or not the knowledge base was modified in the current loop in order to stop inferring. Since 1000 employees in the office required 2.5 person per one location, there were too many instances matching the *domain membership* rules. These instances then required too many SQL calls, which in turn lead to the large performance delay.

7.3 Demonstration Web Client

We have additionally created an *eye-catching* web client to allow an human-based interface to *Me-Centric Domain Server*. We show a snapshot of the web client in Figure 7.2.

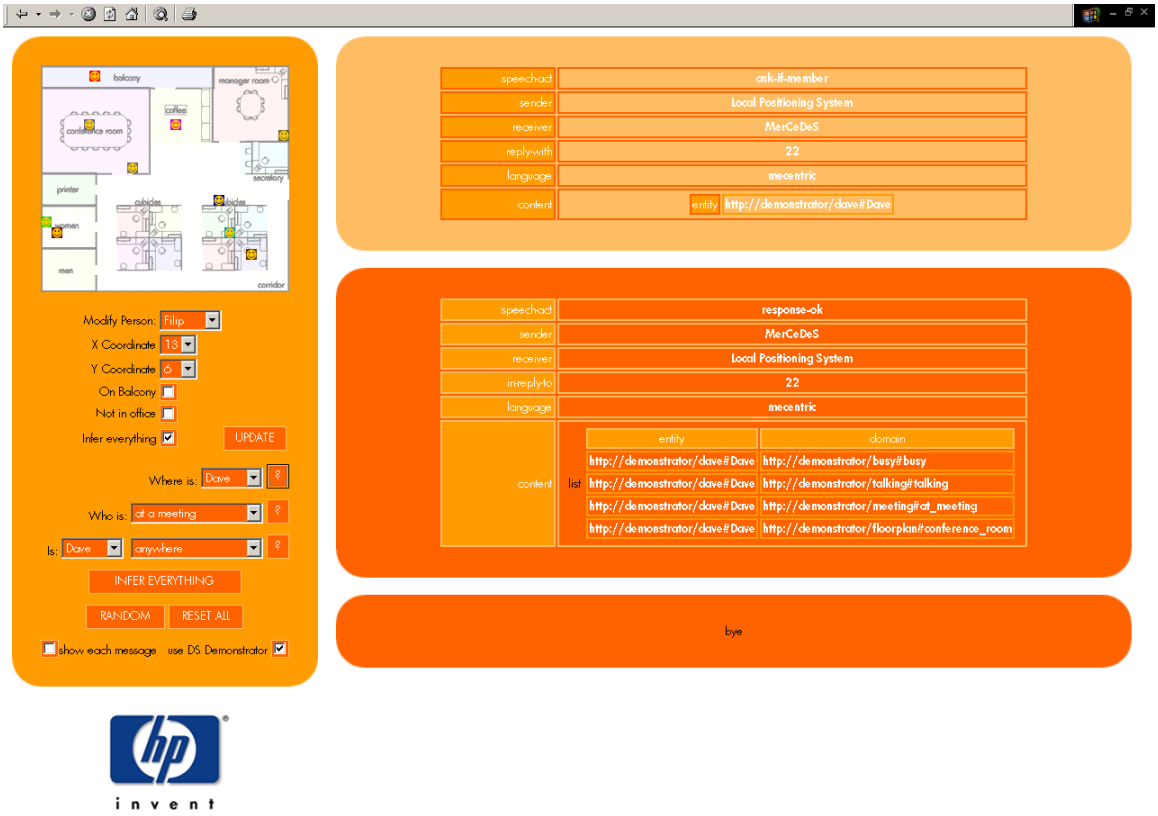


Figure 7.2: Me-Centric Domain Server Demonstrator Web Client.

For this web client, we emphasize that we have not modified the code on the server side at all. Instead, we have only created an HTML page, which relies heavily on JavaScript for providing both the GUI of the client as well as for generating *Me-Centric* messages according to the specification from Section 5.5.1. This web client again uses the same office environment with the same definitions of *domains*. Additionally, the web client defines 10 imaginative employees: Alice, Bob, Carol, Dave, Eve, Filip, George, Hugh, Ian, and John. We have randomly chosen Eve to be the office secretary and Filip as the manager.

The interface allows its user to modify the reading of any sensor. The user can trigger the balcony door sensor to send a message to *Me-Centric Domain Server* that a particular employee stepped out onto the balcony. Similarly, the user can trigger the main entrance sensor as well as assign any coordinate to any employee. Additionally, the web client can automatically randomize all locations for all employees to better

illustrate the *Me-Centric Domain Server* capabilities.

Additionally, the web client provides a human-readable interface for querying the *Me-Centric Domain Server*. Users can ask question like "Where is Dave?" and "Who is at a meeting?" Users can also ask specific questions, such as "Is Dave at a meeting?" These questions are then converted into their appropriate message representations, which are optionally shown to the user for approval and then send to the *Me-Centric Domain Server* using the HTTP POST method.

Upon receipt of any request, the *Me-Centric Domain Server* modifies and/or queries its knowledge base and returns the corresponding answer. The user can either receive the exact representation of the answer the *Me-Centric Domain Server* is sending or it can be optionally converted into a *pretty-print* HTML representation.

Chapter 8

Conclusions

In this report, we have described the need for a generic service that is able to collect static and dynamic information. The service must reason over the collected information and deduce all possible contextual knowledge. Next, the service must also provide a generic querying interface to allow other applications, appliances, and/or agents to utilize the knowledge in customizing their services.

We have defined an *entity* to represent any object or abstract term whose context we wish to know. We then defined the relavant context of each entity in terms of *domains*. Each *domain* represents a small micro-world. It includes a list of policies and roles that are appropriate in that micro-world, and it also defines *membership requirement* functions. These are used to determine whether or not an *entity* is a member of this *domain*.

We have argued for the use of a semantically rich language to describe the context information, and have selected RDF(S) for our implementation. We have created ontologies for describing *entities*, *domains* as well as for expressing *domain requirement* functions. This allowed us to infer additional implicit information that would be impossible with the use of a syntax-level language only.

We have then designed and implemented *Me-Centric Domain Server* as a service for aggregating and interpreting contextual information. This service relies on the use of semantically-rich metadata and uses a forward-chaining principle for inferring additional knowledge. We have utilized a relational database as the

underlying storage, which provides a quick response for modifying the knowledge base. At the same time, the current implementation requires an additional improvements to overcome the difficulties of asserting and retracting implicit knowledge. We believe that it can be improved by creating custom SQL procedures instead of straight SQL queries; however, this was not in the scope of the current project.

Appendix A

RDF to SQL query conversion

An example of an rule used to define when an *entity* is member of the *meeting-with-manager domain*.

```
(and (or (and (eval 'http://demonstrator/meeting#variable1'
                    'http://mecentric.hpl.hp.com/mecentric#isMemberOf'
                    'http://demonstrator/floorplan#conference_room'
              )
              (and (eval 'http://demonstrator/meeting#variable2'
                         'http://mecentric.hpl.hp.com/mecentric#isMemberOf'
                         'http://demonstrator/floorplan#conference_room'
                   )
                   (eval 'http://demonstrator/meeting#variable1'
                         'http://mecentric.hpl.hp.com/mecentric#vNotEqual'
                         'http://demonstrator/meeting#variable2
                   )))
         (and (eval 'http://demonstrator/meeting#variable1'
                    'http://mecentric.hpl.hp.com/mecentric#isMemberOf'
                    'http://demonstrator/floorplan#manager_room'
              )
              (and (eval 'http://demonstrator/meeting#variable2'
                         'http://mecentric.hpl.hp.com/mecentric#isMemberOf'
                         'http://demonstrator/floorplan#manager_room'
                   )
                   (eval 'http://demonstrator/meeting#variable1'
                          'http://mecentric.hpl.hp.com/mecentric#vNotEqual'
                          'http://demonstrator/meeting#variable2'
                   ))))
     (eval 'http://demonstrator/meeting#variable2'
           'http://www.w3.org/1999/02/22-rdf-syntax-ns#type'
           'http://demonstrator/person#Manager'
     ))
```

The same example encoded using RDF(S).

```
<m:Domain rdf:about="&meeting;at_meeting_with_manager">
 <m:dmembershipRequirement>
  <m:KBIfThenCondition>
   <m:if>
    <m:ANDFunction>
     <m:first>
      <m:ORFunction>
```

```
      <m:first rdf:resource="&meeting;inconference_room" />
      <m:rest rdf:resource="&meeting;inmanager_room" />
     </m:ORFunction>
    </m:first>
    <m:rest>
     <m:VerifyFunction>
      <m:verify>
       <m:Statement>
        <m:S rdf:resource="&meeting;variable2" />
        <m:P rdf:resource="&rdf;type" />
        <m:O rdf:resource="&p;Manager" />
       </m:Statement>
      </m:verify>
     </m:VerifyFunction>
    </m:rest>
   </m:ANDFunction>
  </m:if>
  <m:then>
   <m:KBAssertAction>
    <m:add>
     <m:Statement>
      <m:S rdf:resource="&meeting;variable1" />
      <m:P rdf:resource="&m;isMemberOf" />
      <m:O rdf:resource="&meeting;at_meeting_with_manager" />
     </m:Statement>
    </m:add>
   </m:KBAssertAction>
  </m:then>
 </m:KBIfThenCondition>
</m:dmembershipRequirement>
</m:Domain>

<m:ANDFunction rdf:about="&meeting;inconference_room">
 <m:first>
  <m:VerifyFunction>
   <m:verify>
    <m:Statement>
     <m:S rdf:resource="&meeting;variable1" />
     <m:P rdf:resource="&m;isMemberOf" />
     <m:O rdf:resource="&f;conference_room" />
    </m:Statement>
   </m:verify>
  </m:VerifyFunction>
 </m:first>
 <m:rest>
  <m:ANDFunction>
   <m:first>
    <m:VerifyFunction>
     <m:verify>
      <m:Statement>
       <m:S rdf:resource="&meeting;variable2" />
       <m:P rdf:resource="&m;isMemberOf" />
       <m:O rdf:resource="&f;conference_room" />
      </m:Statement>
     </m:verify>
```

```
    </m:VerifyFunction>
   </m:first>
   <m:rest>
    <m:VerifyFunction>
     <m:verify>
      <m:Statement>
       <m:S rdf:resource="&meeting;variable1" />
       <m:P rdf:resource="&m;vNotEqual" />
       <m:O rdf:resource="&meeting;variable2" />
      </m:Statement>
     </m:verify>
    </m:VerifyFunction>
   </m:rest>
  </m:ANDFunction>
 </m:rest>
</m:ANDFunction>

<m:ANDFunction rdf:about="&meeting;inmanager_room">
 <m:first>
  <m:VerifyFunction>
   <m:verify>
    <m:Statement>
     <m:S rdf:resource="&meeting;variable1" />
     <m:P rdf:resource="&m;isMemberOf" />
     <m:O rdf:resource="&f;manager_room" />
    </m:Statement>
   </m:verify>
  </m:VerifyFunction>
 </m:first>
 <m:rest>
  <m:ANDFunction>
   <m:first>
    <m:VerifyFunction>
     <m:verify>
      <m:Statement>
       <m:S rdf:resource="&meeting;variable2" />
       <m:P rdf:resource="&m;isMemberOf" />
       <m:O rdf:resource="&f;manager_room" />
      </m:Statement>
     </m:verify>
    </m:VerifyFunction>
   </m:first>
   <m:rest>
    <m:VerifyFunction>
     <m:verify>
      <m:Statement>
       <m:S rdf:resource="&meeting;variable1" />
       <m:P rdf:resource="&m;vNotEqual" />
       <m:O rdf:resource="&meeting;variable2" />
      </m:Statement>
     </m:verify>
    </m:VerifyFunction>
   </m:rest>
  </m:ANDFunction>
 </m:rest>
```

```
</m:ANDFunction>
```

The same rule converts into these two following SQL queries (due to the *ORFunction*).

```
SELECT
       t1.S as _v1,t2.S as _v2
FROM
       rdf_modelspo as t1,rdf_modelspo as t2,rdf_modelspo as t3
WHERE
      (t1.P='http://mecentric.hpl.hp.com/mecentric#isMemberOf' &&
       t1.O='http://demonstrator/floorplan#conference_room' &&
       t2.P='http://mecentric.hpl.hp.com/mecentric#isMemberOf' &&
       t2.O='http://demonstrator/floorplan#conference_room' &&
       t1.S<>t2.S &&
       t3.S=t2.S &&
       t3.P='http://www.w3.org/1999/02/22-rdf-syntax-ns#type' &&
       t3.O='http://demonstrator/person#Manager')
```

and

```
SELECT
       t1.S as _v1,t2.S as _v2
FROM
       rdf_modelspo as t1,rdf_modelspo as t2,rdf_modelspo as t3
WHERE
      (t1.P='http://mecentric.hpl.hp.com/mecentric#isMemberOf' &&
       t1.O='http://demonstrator/floorplan#manager_room' &&
       t2.P='http://mecentric.hpl.hp.com/mecentric#isMemberOf' &&
       t2.O='http://demonstrator/floorplan#manager_room' &&
       t1.S<>t2.S &&
       t3.S=t2.S &&
       t3.P='http://www.w3.org/1999/02/22-rdf-syntax-ns#type' &&
       t3.O='http://demonstrator/person#Manager')
```

Appendix B

Me-Centric Domain Server Ontology

```
<?xml version='1.0' encoding='ISO-8859-1'?>

<!DOCTYPE rdf:RDF [
 <!ENTITY rdf 'http://www.w3.org/1999/02/22-rdf-syntax-ns#'>
 <!ENTITY mecentric 'http://mecentric.hpl.hp.com/mecentric#'>
 <!ENTITY rdfs 'http://www.w3.org/TR/1999/PR-rdf-schema-19990303#'>
]>

<rdf:RDF xmlns:rdf="&rdf;"
 xmlns:rdfs="&rdfs;"
 xmlns:mecentric="&mecentric;">

<rdfs:Class rdf:about="&mecentric;ANDFunction"
 rdfs:label="ANDFunction">
 <rdfs:subClassOf rdf:resource="&mecentric;BooleanFunction"/>
</rdfs:Class>

<rdfs:Class rdf:about="&mecentric;BooleanFunction"
 rdfs:label="BooleanFunction">
 <rdfs:subClassOf rdf:resource="&mecentric;VFunction"/>
</rdfs:Class>

<rdfs:Class rdf:about="&mecentric;Domain"
 rdfs:label="Domain">
 <rdfs:subClassOf rdf:resource="&rdfs;Resource"/>
</rdfs:Class>

<rdfs:Class rdf:about="&mecentric;Entity"
 rdfs:label="Entity">
 <rdfs:subClassOf rdf:resource="&rdfs;Resource"/>
</rdfs:Class>

<rdfs:Class rdf:about="&mecentric;KBAction"
 rdfs:label="KBAction">
 <rdfs:subClassOf rdf:resource="&rdfs;Resource"/>
</rdfs:Class>

<rdfs:Class rdf:about="&mecentric;KBAssertAction"
 rdfs:label="KBAssertAction">
```

```
 <rdfs:subClassOf rdf:resource="&mecentric;KBAction"/>
</rdfs:Class>

<rdfs:Class rdf:about="&mecentric;KBDisassertAction"
 rdfs:label="KBDisassertAction">
 <rdfs:subClassOf rdf:resource="&mecentric;KBAction"/>
</rdfs:Class>

<rdfs:Class rdf:about="&mecentric;KBIfThenCondition"
 rdfs:label="KBIfThenCondition">
 <rdfs:subClassOf rdf:resource="&rdfs;Resource"/>
</rdfs:Class>

<rdf:Property rdf:about="&mecentric;O"
 rdfs:label="O">
 <rdfs:domain rdf:resource="&mecentric;Statement"/>
 <rdfs:range rdf:resource="&rdfs;Resource"/>
</rdf:Property>

<rdfs:Class rdf:about="&mecentric;ORFunction"
 rdfs:label="ORFunction">
 <rdfs:subClassOf rdf:resource="&mecentric;BooleanFunction"/>
</rdfs:Class>

<rdf:Property rdf:about="&mecentric;P"
 rdfs:label="P">
 <rdfs:domain rdf:resource="&mecentric;Statement"/>
 <rdfs:range rdf:resource="&rdfs;Resource"/>
</rdf:Property>

<rdf:Property rdf:about="&mecentric;S"
 rdfs:label="S">
 <rdfs:domain rdf:resource="&mecentric;Statement"/>
 <rdfs:range rdf:resource="&rdfs;Resource"/>
</rdf:Property>

<rdfs:Class rdf:about="&mecentric;Statement"
 rdfs:label="Statement">
 <rdfs:subClassOf rdf:resource="&rdfs;Resource"/>
</rdfs:Class>

<rdfs:Class rdf:about="&mecentric;VFunction"
 rdfs:label="VFunction">
 <rdfs:subClassOf rdf:resource="&rdfs;Resource"/>
</rdfs:Class>

<rdfs:Class rdf:about="&mecentric;Variable"
 rdfs:label="Variable">
 <rdfs:subClassOf rdf:resource="&mecentric;Entity"/>
</rdfs:Class>

<rdfs:Class rdf:about="&mecentric;VerifyFunction"
 rdfs:label="VerifyFunction">
 <rdfs:subClassOf rdf:resource="&mecentric;VFunction"/>
</rdfs:Class>
```

```
<rdf:Property rdf:about="&mecentric;vEqual"
 rdfs:label="equal to">
</rdf:Property>

<rdf:Property rdf:about="&mecentric;vGreater"
 rdfs:label="greater than">
</rdf:Property>

<rdf:Property rdf:about="&mecentric;vGreaterOrEqual"
 rdfs:label="greater than or equal to">
</rdf:Property>

<rdf:Property rdf:about="&mecentric;vLess"
 rdfs:label="less than">
</rdf:Property>

<rdf:Property rdf:about="&mecentric;vLessOrEqual"
 rdfs:label="less than or equal to">
</rdf:Property>

<rdf:Property rdf:about="&mecentric;vNotEqual"
 rdfs:label="not equal">
</rdf:Property>

<rdf:Property rdf:about="&mecentric;add"
 rdfs:label="add">
 <rdfs:domain rdf:resource="&mecentric;KBAssertAction"/>
 <rdfs:range rdf:resource="&mecentric;Statement"/>
</rdf:Property>

<rdf:Property rdf:about="&mecentric;dDisjointWith"
 rdfs:label="dDisjointWith">
 <rdfs:range rdf:resource="&mecentric;Domain"/>
 <rdfs:domain rdf:resource="&mecentric;Domain"/>
</rdf:Property>

<rdf:Property rdf:about="&mecentric;dEqualTo"
 rdfs:label="dEqualTo">
 <rdfs:domain rdf:resource="&mecentric;Domain"/>
 <rdfs:range rdf:resource="&mecentric;Domain"/>
</rdf:Property>

<rdf:Property rdf:about="&mecentric;dIntersectionOf"
 rdfs:label="dIntersectionOf">
 <rdfs:domain rdf:resource="&mecentric;Domain"/>
 <rdfs:range rdf:resource="&rdfs;Resource"/>
</rdf:Property>

<rdf:Property rdf:about="&mecentric;dSubsetOf"
 rdfs:label="dSubsetOf">
 <rdfs:range rdf:resource="&mecentric;Domain"/>
 <rdfs:domain rdf:resource="&mecentric;Domain"/>
</rdf:Property>
```

```
<rdf:Property rdf:about="&mecentric;dUnionOf"
 rdfs:label="dUnionOf">
 <rdfs:domain rdf:resource="&mecentric;Domain"/>
 <rdfs:range rdf:resource="&rdfs;Resource"/>
</rdf:Property>

<rdf:Property rdf:about="&mecentric;description"
 rdfs:label="description">
 <rdfs:domain rdf:resource="&mecentric;Domain"/>
 <rdfs:range rdf:resource="&rdfs;Resource"/>
</rdf:Property>

<rdf:Property rdf:about="&mecentric;dmembershipRequierement"
 rdfs:label="dmembershipRequierement">
 <rdfs:domain rdf:resource="&mecentric;Domain"/>
 <rdfs:range rdf:resource="&mecentric;KBIfThenCondition"/>
</rdf:Property>

<rdf:Property rdf:about="&mecentric;domainPolicy"
 rdfs:label="domainPolicy">
 <rdfs:domain rdf:resource="&mecentric;Domain"/>
 <rdfs:range rdf:resource="&rdfs;Resource"/>
</rdf:Property>

<rdf:Property rdf:about="&mecentric;first"
 rdfs:label="first">
 <rdfs:domain rdf:resource="&mecentric;BooleanFunction"/>
 <rdfs:range rdf:resource="&mecentric;VFunction"/>
</rdf:Property>

<rdf:Property rdf:about="&mecentric;hasMember"
 rdfs:label="hasMember">
 <rdfs:domain rdf:resource="&mecentric;Domain"/>
 <rdfs:range rdf:resource="&mecentric;Entity"/>
</rdf:Property>

<rdf:Property rdf:about="&mecentric;if"
 rdfs:label="if">
 <rdfs:domain rdf:resource="&mecentric;KBIfThenCondition"/>
 <rdfs:range rdf:resource="&mecentric;VFunction"/>
</rdf:Property>

<rdf:Property rdf:about="&mecentric;isMemberOf"
 rdfs:label="isMemberOf">
 <rdfs:range rdf:resource="&mecentric;Domain"/>
 <rdfs:domain rdf:resource="&mecentric;Entity"/>
</rdf:Property>

<rdf:Property rdf:about="&mecentric;remove"
 rdfs:label="remove">
 <rdfs:domain rdf:resource="&mecentric;KBDisassertAction"/>
 <rdfs:range rdf:resource="&mecentric;Statement"/>
</rdf:Property>

<rdf:Property rdf:about="&mecentric;rest"
```

```
 rdfs:label="rest">
 <rdfs:domain rdf:resource="&mecentric;BooleanFunction"/>
 <rdfs:range rdf:resource="&mecentric;VFunction"/>
</rdf:Property>

<rdf:Property rdf:about="&mecentric;then"
 rdfs:label="then">
 <rdfs:range rdf:resource="&mecentric;KBAction"/>
 <rdfs:domain rdf:resource="&mecentric;KBIfThenCondition"/>
</rdf:Property>

<rdf:Property rdf:about="&mecentric;verify"
 rdfs:label="verify">
 <rdfs:range rdf:resource="&mecentric;Statement"/>
 <rdfs:domain rdf:resource="&mecentric;VerifyFunction"/>
</rdf:Property>

</rdf:RDF>
```

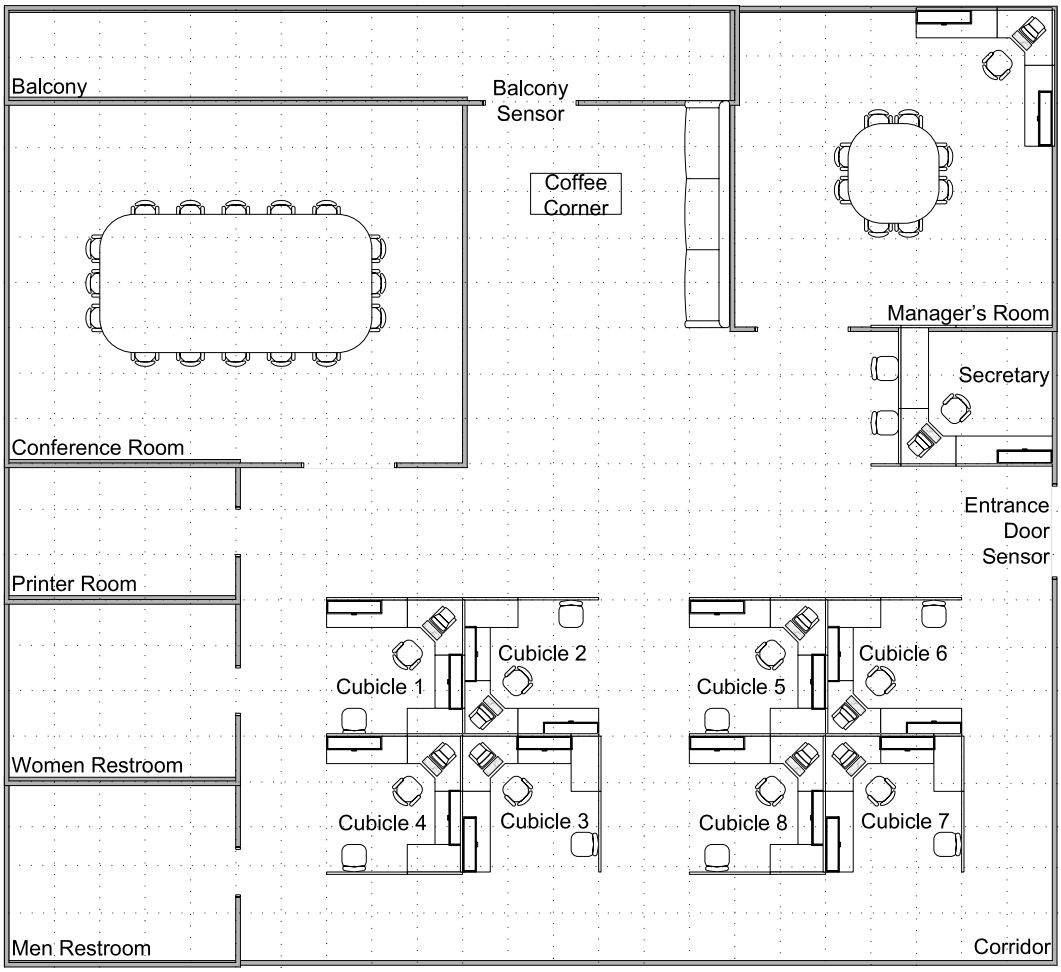
Appendix C

Me-Centric Message Structure

```
<message> =   ( :speech-act "<speechact>"
                :sender "<sender>"
                :receiver "<receiver>"
                :date "<date>"
                :language "<language>"
                :reply-with "<reply-with>"
                :in-reply-to "<reply-with>"
                :content <item> | <list> )

<speechact> =   add-domain | add-entity | add-membership-information | ask-if-member |
                infer-domain-members | infer-entity-memberships | remove-domain |
                remove-domain-members | remove-entity | remove-entity-memberships |
                remove-membership-information | update-domain | update-entity |
                response | response-ok | response-error

<sender> =      {SENDER_ID}

<receiver> =    {RECEIVER_ID}

<date> =        {DATE}

<language> =    mecentric

<reply-with> =  {REPLY_MSG_ID}

<in-reply-to> = {REPLY_MSG_ID}

<item> =        (:<attribute> "<value>" :<attribute> "value" ...)

<list> =        (:list (<item>,<item>,...,<item>))

<attribute> =   {ATTRIBUTE_NAME}

<value> =       {ATTRIBUTE_VALUE}
```

Appendix D

Demonstration Floorplan



Bibliography

[1] MySQL AB. The MySQL relational database server. http://www.mysql.com/.

[2] The Apache Software Foundation. Tomcat Java HTTP Server . http://jakarta.apache.org/tomcat/.

[3] T. Bray, J. Paoli, and C. Sperberg-MacQueen. Extensible Markup Language. http://www.w3.org/TR/1998/REC-xml19980210, 1998.

[4] D. Brickley and R. Guha. Resource Description Framework (RDF) Schema Specification 1.0 - W3C Recommendation. http://www.w3.org/TR/2000/CR-rdfschema-20000327, 2000.

[5] Bernard Burg, Harumi Kuno, Craig Sayers, and Kevin Smathers. Me-centric Position Paper. HP Internal Draft, May 2002.

[6] R. Cost, T. Finin, A. Joshi, Y. Peng, C. Nicholas, I. Soboroff, H. Chen, L. Kagal, F. Perich, Y. Zou, and S. Tolia. ITtalks: A Case Study in the Semantic Web and DAML+OIL. *IEEE Intelligent Systems Special Issue*, 17(1):40–47, January/February 2002.

[7] Anind K. Dey. *Providing Architectural Support for Building Context-Aware Applications.* PhD thesis, College of Computing, Georgia Institute of Technology, December 2000.

[8] Anind K. Dey and Gregory D. Abowd. Towards a better understanding of context and context-awareness. In *Workshop on the What, Who, Where, When and How of Context-Awareness, affiliated with the CHI 2000*, 2000.

[9] Anind K. Dey, Gregory D. Abowd, and Daniel Salber. A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications. *anchor article of a special issue on Context-Aware Computing, Human-Computer Interaction (HCI) Journal*, pages 97–166, 2001.

[10] Jonathan Dale (ed.). Advanced Web Services. Confidental.

[11] FIPA. FIPA ACL Message Representation in String Specification. http://www.fipa.org, 2001. XC00070G.

[12] Andy Harter, Andy Hopper, Pete Steggles, Andy Ward, and Paul Webster. The Anatomy of a Context-Aware Application. In *Wireless Networks*, 2001.

[13] Tim Kindberg, John Barton, Jeff Morgan, Gene Becker, Debbie Caswell, Philippe Debaty, Gita Gopal, Marcos Frid, Venky Krishnan, Howard Morris, John Schettino, and Bill Serra. People, Places, Things: Web Presence for the Real World. http://cooltown.hp.com/dev/, 2000.

[14] O. Lassila and R. Swick. Resource Description Framework. http://www.w3.org/TR/1999/REC/rdf-syntax-19990222, 1999.

[15] Brian McBride. Jena: Implementing the RDF Model and Syntax Specification. http://www-uk.hpl.hp.com/people/bwm/papers/20001221-paper/, December 2000.

[16] Nuria Oliver. Learning and Inferring Office Activity from Multiple Sensory Streams. Submitted to a conference, 2002.

[17] Nissanka B. Priyantha, Anit Chakraborty, and Hari Balakrishnan. The Cricket location-support system. In *Mobile Computing and Networking*, pages 32–43, 2000.

[18] Open Directory Project. http://dmoz.org/.

[19] Olga Ratsimor, Vladimir Korolev, Anupam Joshi, and Timothy Finin. Agents2Go: An Infrastructure for Location-Dependent Service Discovery in the Mobile Electronic Commerce Environment. In *ACM Mobile Commerce Workshop*, July 2001.

[20] Sleepycat. Berkeley DB - Open Source Embedded Database System. http://sleepycat.com/.