A Context-Oriented RDF Database

Mohammad-Reza Tazari

Computer Graphics Center, Dept. Mobile Information Visualization
Fraunhoferstraße 5, 64283 Darmstadt, Germany
Saied.Tazari@zgdv.de

Abstract. The importance of contextual knowledge in knowledge management and organizational memory is shown in topical literature. Even in an initial visionary scenario for the Semantic Web, one can immediately encounter the contextual knowledge needed to realize the necessary services. Hence, it is not inappropriate to claim that context management is an integral service of the Semantic Web. After discussing the distributed nature of contextual knowledge, we define some requirements for a context-oriented database service and then introduce CORD as a service satisfying those requirements based on the Semantic Web technologies. Selected features of CORD that provide some contribution to the discussions within the Semantic Web research community, like embedded resources, query language, and definition of rules, are discussed in some detail.

1 Introduction

Most of the Semantic Web applications will be context-aware and personalized services. A superficial look at the visionary scenario for explaining some of the features of the Semantic Web in [2] shows the correctness of this claim. Already in the first two sentences of this scenario[1], the following contextual knowledge must be available to realize the service:

- User location (here, Pete's location)
- Setup of the location (to identify devices near Pete and services offered there)
- States of the resources (which of the sound-making devices are "on", the phone "ringing", the phone in "talk" state)
- Characteristics and capabilities of resources (which services can operate precisely those sound-making devices near Pete that are on and loud)

Although the term *context* has a common meaning in the Semantic Web community – see, for example, the definition by Tim Berners-Lee under http://www.w3.org/2000/10/swap/doc/Glossary – we are purposing here a special user-centric view of context. That is, by context we mean the user context in terms of personal, environmental, and

[1] "The entertainment system was belting out the Beatles' "We Can Work It Out" when the phone rang. When Pete answered, his phone turned the sound down by sending a message to all the other local devices that had a volume control." [2]

temporal conditions surrounding him or her. This is the situational view to the context as it is investigated in Mobile Computing and Ubiquitous/Pervasive Computing, too. The whole imaginary scenario in [2] is full of assumptions about the existence of such contextual knowledge.

As already stated in [5], knowledge about the user context is highly distributed. Except for the "current time" that in fact belongs to the user context in diverse forms[2], but has nothing to do with the distribution aspect of the contextual knowledge, most of the other parts of this knowledge can be classified as follows (see also figure 1):

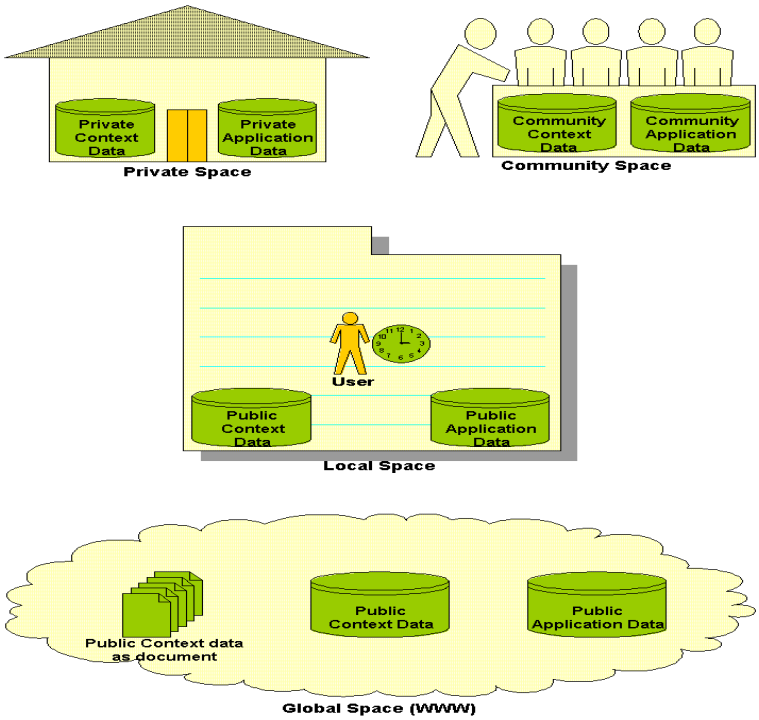

Fig. 1. Distribution of contextual knowledge. At any given time, the user finds himself at a specific location and may be able to use resources there. Nevertheless, he may also be able to use resources from his private space, community spaces to which he belongs, and public resources available globally

– Knowledge directly bound to the user and captured in diverse logical units, such as the user profile[3], profiles of user's mobile and accompanying devices, application

[2] E.g. the absolute time, hour of the day, am / pm, day of the week, etc.

[3] We believe that most parts of the user context may be imparted in form of profiles and define a *profile* as *the storage unit for a coherent collection of key-value pairs describing a distinct resource, location, or user*. If the described resource exists in an electronic form, then its profile provides the corresponding metadata; otherwise a profile may simultaneously serve as the electronic representation of the resource itself.

data from the domain of personal information management (i.e. PIM data, such as to-dos, appointments, contacts), application-specific user preferences[4], etc.
- Knowledge bound to the user's location and captured as the location profile and profiles of resources available there, where the location profile serves as an integrating unit for all other info units.
- Knowledge bound to communities (to which the user belongs) and captured as group-based defaults, profiles of shared resources, and shared application data.
- Public knowledge independent of the user, communities, and locations that will be made available through the Web, such as profiles of public resources (e.g. services that can be utilized by all, independent of the locations of the two ends[5]) and profiles of classes of resources, which provide default values for a set of concrete resources.

Obviously, the contextual knowledge includes many shared units, such as the user profile, the location profile, and the profiles of several resources, of which different context-aware applications may make use. Hence, a standardized service is needed for managing profiles and offering shared mechanisms, relieving context-aware applications from certain common overheads like monitoring the user context and recognizing interesting situations. We call such a service the *context management service*. In [5], we discussed the requirements for a context management service and in [15], the aspect of modeling user context. Here, we focus on the data management aspects of this service.

1.1 Requirements for a Context-Oriented Database Service

In the discussion above, we have emphasized three specific points having to do with data management aspects of the context management service: data distribution, organizing data in profiles, and support for default values (group-based or class-based defaults). The first aspect leads us to the requirement that a context-oriented database service (CODBS) must overcome the problem with the distribution of contextual knowledge. Secondly, if profiles as a collection of key-value pairs are the storage units of a CODBS, then it must support arbitrarily structured keys and values[6]. Thirdly, support for default values would mean that there must be a mechanism for profiles of more concrete resources to inherit data from profiles of related, albeit more abstract, resources. All of the above actually reveal different aspects of data organization, namely data organization within a profile, between profiles, and between databases.

To specify further requirements, we must zoom in on the data organization within a profile. The organization of data within a profile will primarily be reflected in its keys. That is, a fundamental requirement is the possibility of expressing complex

[4] Context-aware and personalized applications may have some personalization scheme that is specific to them and hence must be managed separately from the user profile that is a shared unit based on a shared ontology that models user profiles in general.

[5] The locality of the resource may eventually play an important role in order for it to be selected / referenced / used from among all competing resources.

[6] This requirement is refined further in the next paragraph.

structures via keys. Another aspect, however, has to do with the values. Values may be literal, which raises the question about support for data types, or references to other resources. A key may be associated with a single value or with more than one value. The latter case leads to the support for sequences, bags, and sets of alternative values. Values may be valid only for a specific time period[7] or independent of time. Last but not least, they may be conditional/situational, meaning that a key may be associated with different alternative values for different situations.

Assuming that for each profile type there is a schema defining its structure and asserting some statements about its semantics, a CODBS must also use schemas in order to be able to ensure data integrity by accepting data that is in accordance with the structural and type-related assertions made in the schema. In addition, context-aware applications will be able to ask for the underlying schemas if they are not able to interpret some contextual knowledge.

Finally, a CODBS must provide a triggering mechanism for catching database events, because changes in the state of the contextual knowledge may influence the situation in which the user finds himself. The transition from one situation to another is an important event for context-aware applications.

Hence, we can summarize the requirements for a CODBS, as follows:

1. Managing profiles in accordance with their underlying schemas and guaranteeing data integrity based on the assertions made in the schemas
2. Providing a centralized view of the highly distributed contextual knowledge
3. Providing a triggering mechanism depending on complex situational DB events
4. Support for conditional values
5. Support for defining hierarchies of profiles that share the same schema to facilitate the automatic inheritance of default values
6. Support for expressing complex structures via keys within profiles
7. Support for literal values with different data types
8. Support for sequences, bags, and sets of alternative values
9. Support for temporary values [13]
10. Support for using references to other resources as values

2 CORD: The Context-Oriented RDF Database

We have developed an RDF database called CORD that is the foundation for our context management service. The context-manager itself is the wrapper agent that provides an interface for agent communication [5]. CORD implements most of the features enumerated as requirements for a CODBS in 1.1. After justifying the basic approach, we discuss in the following subsections those aspects of our solution that provide some contribution to the discussions within the Semantic Web research community.

[7] Especially sensory data may be of a temporary nature.

2.1 Choosing the Semantic Web Technology

Obviously, the exchange of contextual knowledge must be based on a knowledge representation paradigm. On the other hand, profiles are nothing other than descriptions about distinct resources. These two statements alone, along with the fact that RDF provides solid concepts for not only describing resources, but also for modeling them, justify the selection of RDF, RDF schema, and OWL. Besides, the XML syntax of RDF fit perfectly into our multi-agent system, where XML was the content language of choice in agent communication messages.

2.2 Why a New Database Service?

Many of the projects dealing with RDF data stores use a relational DBMS (see, for example, the two surveys in [1] and [10] summarizing some of them). A general-purpose mapping of the RDF data model onto the relational model, where no assumptions about the type of resources being described are made, leads to the definition of few tables with few columns (see, for example, proposed DB schemas at [11]). Basically, if we consider the RDF data model as a set of triples, a three-column table will come up with a huge number of rows storing the statements, each with a subject, a predicate, and an object. Even if we consider the RDF data model as a directed, labeled graph, the relational database design will come up with similar results. With such a modeling, answering queries about complex resources may lead to many self-joins on one big table – depending on the entry point given by the query – where the consequences for the performance are not known.

Choosing an object-oriented database management system would not change the above situation, either. The issue is: relational or object-oriented DBMSs may meaningfully be used where a specific domain with concrete entity types is being modeled. That is, if you know the types of resources being described in your RDF data store, then you can provide a conventional database design with a meaningful database schema. The database schema would then reflect at least parts of what you state in the RDF/OWL schema for modeling the same resource types. A wrapper could then provide the knowledge stored in the database in terms of RDF statements to the world outside.

Due to the fact that the context management solution must be open for managing profiles of resources having arbitrary types, choosing a relational or object-oriented DBMS would confront us with the same dilemma as described in the previous paragraphs. On the other hand, a glance at our requirements, especially the requirements #3, #4, #5, and #9, shows that an existing database service may hardly satisfy all of them. Although most of the DBMSs do provide a triggering mechanism, even the utilization of stored procedures in the domain of relational databases or the class methods in the domain of object-oriented databases is no solution for the efficient recognition of interesting situations, that may be defined using complex conditions[8]. The main reason is that the situations to be recognized are not definable all at once, but their definitions will be added and removed dynamically. For the

[8] Cp. also the discussion in section 2.5.

traditional database services, this would mean dynamism at the schema level. The concept of conditional values, discussed in section 2.5, is new and no direct support could be found in the domain of relational databases for storing them in arbitrary columns of arbitrary rows of tables. The methods in Object-oriented databases do not solve the problem, either, because they are defined within classes and are the same for all instances. The automatic inheritance of default values requires hierarchical relationship between rows of tables or instances of classes, which is not given, either. Finally, support for temporary values presupposes a timeline management for each column of each row or each field of each instance, which is not supported by traditional DBMSs.

For the purpose of profile management, we tried to provide the OWL schemas for user profiles, location profiles, terminal profiles, service profiles, and agent profiles (as a special case for service-offering software components) [15, 16]. Not only the term "profile management", but also the complex structure of the above-enumerated profile types, the requirement for inheriting default values from more abstract profile instances in more concrete profile instances, and the concrete use cases in our projects caused us to choose profiles as our storage units. Having to meet the requirements listed in section 1.1, choosing profiles as the storage unit, having to manage profiles of arbitrary types, and considering the fact that each instance of the context manager deals with *few* instances of *complex* resources caused us to decide in favor of developing CORD.

2.3 Profiles and Their XML-based RDF Representation

A profile is a reusable resource that can be identified via a URI. This URI, which may be given as the value for *xml:base* in the XML representation of the profile, has the following structure:

```
cord://<host>:<port>/profiles/<profile-name>
```

Internally, profiles are implemented as (hashed) trees quite similar to *ldap* or *Windows™ registry*. The main difference with those solutions is the lack of a global root binding all of the (sub-)trees in one big tree integrating them.

As stated before, profiles are containers of key-value pairs. We call each such pair a *context element*, where the key serves both as the URI of the context element and as the source of its semantics. In the tree representation of profiles, however, there is no clear-cut distinction between keys and values. In addition to the leaves of the tree that represent the literal values or URI references, any node in the tree can be seen as a value associated with its path. Then, the path together with the base URI of the profile serves as the key. Except for the root of the tree that represents the whole profile resource (denoted as `cord://<host>:<port>/profiles/<profile-name>#`), all other branch nodes represent some embedded resource identified with a URI of the form `cord://<host>:<port>/profiles/<profile-name>#<path>`. Paths are made of *NDName*s (XML-names[9] minus '.') concatenated by dots ('.'), e.g. `a.b.c` would be a valid path. Each NDName

[9] See http://www.w3.org/TR/REC-xml#NT-Name.

corresponds to a property of the concrete resource addressed thus far. The possibility of using paths as part of keys meets the requirement for expressing complex structures via keys within profiles.

An example will further illustrate the usage of paths in keys. Let's assume that a schema with the URI `http://www.zgdv.de/CORD/schemas/UserProfile` defines, among others, the following concepts:

- the classes *UserProfile*, *PersonalInfo*, and *PersonName*.
- the property *personalInfo* with domain *UserProfile* and range *PersonalInfo*.
- the property *name* having *PersonalInfo* as its domain and *PersonName* as its range.
- the properties *first*, *middle*, *last*, and *nick* having *PersonName* as their domain and *xsd:string* as their range.

Then, the following RDF description represents my profile partially:

Sample 1. Partial RDF representation of a user profile

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns="http://www.zgdv.de/CORD/schemas/UserProfile#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xml:base="cord://st.zgdv.de:999/profiles/me">
  <UserProfile rdf:about="#">
    <personalInfo>
      <PersonalInfo rdf:about="#personalInfo">
        <name>
          <PersonName rdf:about="#personalInfo.name">
            <first>Mohammad-Reza</first>
            <last>Tazari</last>
            <nick>Saied</nick>
          </PersonName>
        </name>
      </PersonalInfo>
    </personalInfo>
  </UserProfile>
</rdf:RDF>
```

This results in the tree representation shown in figure 2 and the set of context elements shown in table 1.

Table1. Set of context elements (key-value pairs) resulting from Sample 1

Key	Value
cord://st.zgdv.de:999/profiles/me#	The whole profile resource
cord://st.zgdv.de:999/profiles/me#rdf:type	http://www.zgdv.de/CORD/schemas/UserProfile#UserProfile
cord://st.zgdv.de:999/profiles/me#personalInfo	The embedded resource rooted at 'personalInfo'
cord://st.zgdv.de:999/profiles/me#personalInfo.rdf:type	http://www.zgdv.de/CORD/schemas/UserProfile#PersonalInfo
cord://st.zgdv.de:999/profiles/me#personalInfo.name	The embedded resource rooted at 'name'
cord://st.zgdv.de:999/profiles/me#personalInfo.name.rdf:type	http://www.zgdv.de/CORD/schemas/UserProfile#PersonName
cord://st.zgdv.de:999/profiles/me#personalInfo.name.first	Mohammad-Reza
cord://st.zgdv.de:999/profiles/me#personalInfo.name.last	Tazari
cord://st.zgdv.de:999/profiles/me#personalInfo.name.nick	Saied

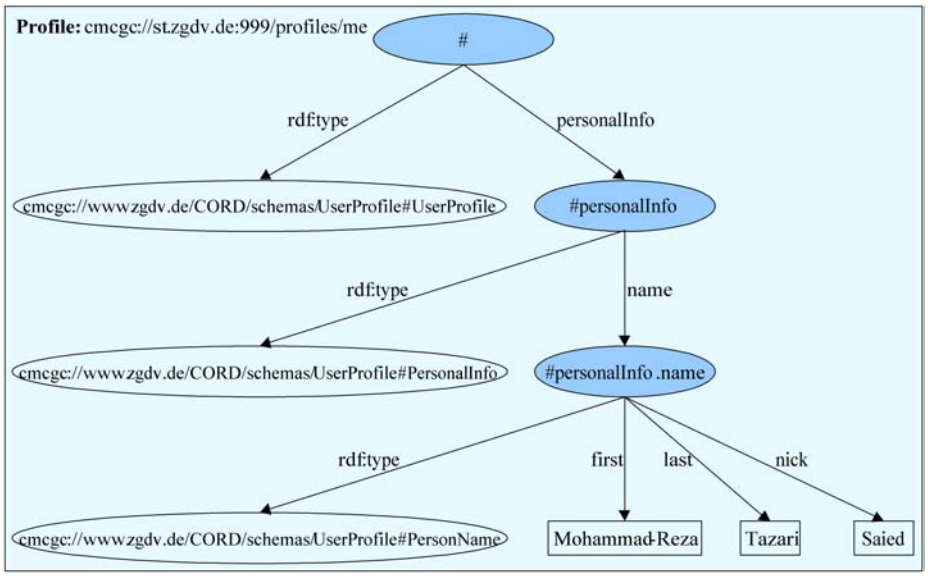

Fig. 2. Internal tree representation resulting from Sample 1 (leaf nodes have no background color)

Embedded Resources. The above model leads to some issues that are not handled in RDF or OWL standards. The most important issue concerns resources embedded in the profile. Each non-leaf node within the tree representation of a profile actually represents such an embedded resource as an identifiable resource.

The embedded resources result from either the *part-of* association – one of the fundamental concepts in object-oriented modeling – or the theorem of *weak entity classes* in database management. In UML, for example, associations having a diamond on one side indicate that the class on that side represents composite objects having instances of the class on the other side as their parts. They go even further and say that if the diamond is darkened, then the instances of the class on the other side may never exist independently from instances of the class on the side of the diamond – quite similar to the concept of weak entity classes in database management.

However, there is currently no way to specify part-of associations as such in RDF schema or OWL. CC/PP [8], as an RDF-based approach for articulation and exchange of contextual knowledge in profiles, has proposed the concept of *component*s that can be seen as a solution for this problem. It was not consistent enough, though. It seems that the understanding of profiles in CC/PP is something like the ".ini"-files in Windows™; i.e. all attributes must appear in some component and no attributes within components may have another component as value. None of these restrictions matched our requirements. This approach ignores the evolution of such config-files into tree structures like Windows™ registry[10]. This is the main reason why we decided to establish the following conventions to enforce our idea of profiles:

[10] [6], another research activity on context management, has similar criticisms on CC/PP.

- A schema modeling a profile type always defines a special class named after the schema itself that serves as the "main class". All other classes defined in the schema are either super/sub-classes of the main class, appear in a (nested) part-of association, or represent some weak entity class. The root of a profile instance always represents an instance of this "main class" or its super/sub-classes.
- All resources contained in a profile instance other than the root of the profile have a path as their ID that results from concatenating (via dots) properties binding them to the root of the profile. Hence, property names may not contain dots.
- Many-valued properties refer to instances of rdf:Alt, rdf:Bag, or rdf:Seq as embedded resources with an ID built up in the same way as stated in the previous bullet. This has two implications: 1) the leaf nodes of a profile are always literal values or URI references to other resources and 2) if the elements of the container are some other embedded resources, then they have an ID resulting from appending a '.rdf:_n' to the ID of the container, where *n* is a decimal integer greater than zero with no leading zeros. This means that the requirement for supporting "sequences, bags, and sets of alternative values" is combined with the requirement for "expressing complex structures via keys within profiles".
- In order to maintain profile boundary, all references to external resources are stored as URI references.
- All arcs that transform the tree representation into a graph are automatically redirected to point to a leaf having the equivalent local URI reference as its value.

2.4 The Query Language

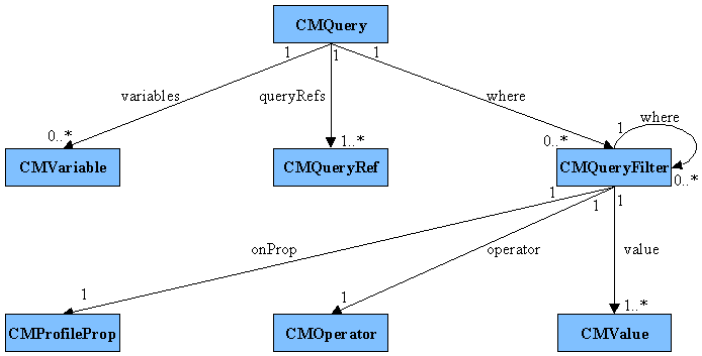

Fig. 3. Concept for CORD queries

Queries submitted to a CORD instance must be RDF descriptions based on the concept summarized in figure 3. Unlike the existing solutions that try to propose a general-purpose RDF query language concentrating on the syntax (see, for example, [7], [9], [12] and [14]), we have concentrated on the application of RDF in describing profiles and our related concepts like paths. Although we are practically using only the XML syntax of RDF to submit queries, deriving a compact SQL-like notation from the same concept is straightforward. In the following subsections, each of the main parts, from which CORD queries are formed, will be discussed in some detail.

Query References. A query must contain at least one query reference. Each query reference will be expanded to a set of matched keys (compare table 1) and an RDF description containing the context elements identified by those keys will be returned as query result. A query reference is a CORD URI-reference with the following wildcarding possibilities (most combinations of them are also legal):

- If the `<host>:<port>` slot is missing, then the CORD instance receiving the query will consolidate all other known CORD instances in the ascertainment of the query results.
- If the profile name is missing, then all profiles managed by the CORD instance will match.
- If the query reference ends with the fragment separator (#), then the whole content of the profile will match.
- If the `<path>` slot begins with a dot (.), then any context element having the remainder of the `<path>` slot as the suffix of its own path will match.
- If the <path> slot ends with a dot (.), then the sub-tree rooted at the resource matching the leading part of the <path> slot will match.
- If the <path> slot contains two subsequent dots (..), then any context element within the sub-tree rooted at the resource matching the leading part of the <path> slot and having the remainder of the <path> slot as the suffix of its path will match.

The special case of `cord://<host>:<port>/profiles/`, where again the `<host>:<port>` slot may be left empty, will cause a query result to contain only a bag of URI references to the matching profiles without the descriptions of their contents.

Variables. A query may have a *sequence* of initialized variables that store literal values or URI references. A subsequent variable may use a previous variable storing a URI reference to store a subordinate value (cp. last paragraph in this section about the usage of variables). This facilitates, among other things, the switching to referenced profiles and inter-profile joins.

Variables may be of type *KeyVariable* or *ValueVariable*. The sub-type influences the interpretation of the value to be assigned to the variable. In the case of key variables, it is expected that the value is a URI reference that must be resolved so that its associated value is assigned to the variable. In the case of value variables, however, the value given will be assigned to the variable as-is, be it a URI reference or not.

There are some predefined variables that are set automatically, normally just before CORD begins to process a new request:

- The current time is stored in variables like *currentTime*, *amPM*, *dayOfMonth*, *dayOfWeek*, etc., quite similar to the constant fields defined in java.util.Calendar.
- The certificate of the agent that sent the request is stored in *accessor*. This is interesting for the definition of conditional values (see section 2.5), when the accessor plays a role in the decision about the applying value.
- As stated before, a query reference will be expanded to a set of matched keys. Each time, after selecting one of the matched keys for further processing, two variables are set automatically that keep their values until CORD leaves the context

of that matched key. These are *currentProfile*, which contains the URI reference of the profile from which the matched context element originates, and *matchedComponent*, which contains the URI reference of the lowest embedded resource matched during the expansion of the query reference into the matched key.

– Two other special variables are set automatically whenever a context element is added, updated, or deleted. These variables are only interesting for the processing of subscriptions (see section 2.5). They are *triggerKey*, which contains the URI reference of the changed element, and *triggerValue*, which contains the new value associated with the key of the context element, if applicable. The setting of *triggerKey* causes the time variables and the *currentProfile* variable to be set automatically in the context of processing the DB event.

In general, whether predefined or defined by the requestor, variables can be used to build up new query or other URI references, can be used in query filters or conditions of rules (see section 2.5), or wherever values are expected. Variable substitution occurs whenever a special construct is found in a way similar to macro expansions in the C programming language. For example, assuming that the standard variable *currentProfile* contains a reference to a user profile in a special context, one can use it in the following construct in place of a direct value:

```
<cord:VarRef>
  <cord:variable rdf:resource="&cord;currentProfile"/>
  <cord:suffix>personalInfo.name.last</cord:suffix>
  <cord:action rdf:resource="&cord;substituteAndEval"/>
</cord:VarRef>
```

If at runtime the variable has the value `cord://st.zgdv.de:999/profiles/me#`, then, due to the specified *action*, the above variable reference is first replaced and expanded to `cord://st.zgdv.de:999/profiles/me#personalInfo.name.last` which will then be evaluated to the literal value `Tazari`. Other possible values for *action* are *substitute* and *evalAndSubstitute*. Another property of the *VarRef* class not used in the above example is *cord:prefix*.

Query Filters. A container of query filters can be used to select only a subset of the matched keys resulting from the expansion of query references. If the container is of type *rdf:Seq*, then an implicit and-connector is assumed between the query filters given in the sequence; in the case of *rdf:*Alt, an implicit or-connector is assumed. Beside query filters, elements of such containers may also be a container of the other type to switch between connector types[11].

A query filter says which criterion must be satisfied in order to keep a previously matched context element in the set of those to be returned in the query result by specifying *what* must be compared *how* with *which value(s)*. To specify the *how*, one must select an operator from the enumeration defined by *CMOperator*. Currently, the

[11] The point with the container type and its relation with the connector type and the possibility of nesting them to switch from one connector type to another is not shown in figure 3 in order to keep the model straightforward.

possible values are *equal*, *greater*, *less*, *in*, *including*, *notEqual*, *notGreater*, *notLess*, *notIn*, and *excluding*.

The criterion must be selected from the enumeration defined by *CMProfileProp* and given as the value (in the form of a URI reference) for a property called *onProp*. The possible values are basically:

- schema: to select context elements coming from a specific profile type. For example, to filter context elements coming from user profiles, one may define a query filter on *schema* property, choose the *equal* operator, and give *http://www.zgdv.de/CORD/schemas/UserProfile* as a single URI reference for the *value* property.
- parents/children: to select context elements from a profile that has the given profiles as its parents/children.
- begin/end/importance/priority[12]: to select context elements whose values are valid during a certain time period (for temporary context values) or satisfy certain weighting criteria (not to be discussed further).
- value: to select context elements whose values satisfy the given condition.
- currentProfile/matchedComponent: combined with a suffix for building up a new query reference, they can be used to filter the matched context elements further. The resulting query reference forms a sub-query with the possibility to check the values returned by the sub-query in the same CMQueryFilter and to further filter them based on the conditions provided by the optional *where* property.

2.5 Rules

CORD supports two forms of rules that are structured similarly: one for forming conditional values and the other one for posting subscription requests. These are discussed in the following subsections.

Conditional Values. Values (especially those given for preferences) can be rule-based, in the sense that the value depends on some contextual state or situation. When a rule-based value is queried, first the cases within the rule will be examined using the current values of referenced context elements. If one of the alternative cases applies, then the associated value is returned, otherwise *rdf:nil*. Figure 4 summarizes the CORD concept for conditional values.

Basically a CORD rule is a "switch-case" construct. Each case has a condition part and a value part. The cases are considered in the *sequence* of their specification. As soon as a case is found whose condition part evaluates to true, the evaluation will cease and the value associated with that case is used as the result of the evaluation. A case without any conditions always evaluates to true. The condition part of each case is a container (of type *rdf:Seq* or *rdf:Alt* quite similar to the containers of query filters – see also footnote 9) of comparisons, where normally values of context elements are compared with literal values or with values of other context elements.

[12] Special properties introduced by CORD and applicable to all nodes within a profile.

Due to some complications in the implementation, conditional values are currently a special case of literal values; this leads to two side effects: 1) the restriction for cases to contain only one value (literal or URI-reference) and 2) the delay in parsing until the conditional value is accessed.

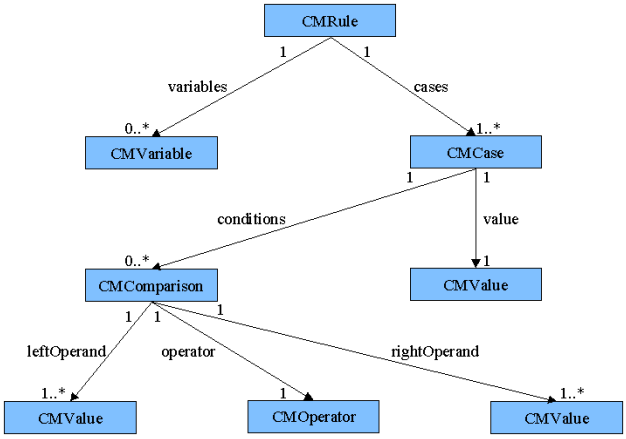

Fig. 4. Concept for conditional values in CORD

From another perspective, we can say that conditional values equip CORD with something like "passive inference". With "passive inference" we mean that on the one hand CORD does not have its own rules to infer the new state of the contextual knowledge, but the logic of inferring comes from "producers"/"providers" of the contextual knowledge. On the other hand, the inferring process only leads to a selection between suggested alternatives depending on the current situation (cp. [5] for our concept of situations as contextual states).

The concept of variables and variable references is already discussed in section 2.4. However, an interesting aspect of using variable references in comparisons is that the corresponding variables don't have to be defined in the variables part of the rule. If they are defined in the rule, they will usually refer to some shared contextual facts (facts known to the CORD instance at the time of rule interpretation). But, since such rules are interpreted at query time, one may define the variables in his or her queries. This way, the facts to be used in the evaluation of the rule may be the premises of the requestor.

Last but not least, a special feature resulting from the inheritance of default values is worth mentioning. Assuming that the profile p_1 is a parent of the profiles p_2 and p_3 and a conditional value defined in p_1 is being inherited by both p_2 and p_3, the use of local URI references in the rule might cause the same rule to return a different value in the context of p_2 compared to the value returned in the context of p_3, even if the two evaluations are performed "simultaneously". An example may illustrate this nice effect better: Assume that a travel planing agent stores a profile for each travel to be planned in the user's personal context-manager letting it inherit from the default travel profile provided by the context-manager of the company where he works. If a

property *transportMeans* in the default profile has a conditional value in the following simplified form

```
if #distance.inKm greater 500
    return my:plane
else if #numberOfCompanions greater 1
    return my:rentACar
else
    return my:train
```

then a query about the transport means for a concrete travel where the user must travel alone to a city 225 km far from his residence would result in `my:train`.

Subscription Requests. Requestors may subscribe for notification or other actions to be performed by CORD. There are two kinds of subscriptions: simple or conditional.

A simple subscription is formed from a bag of query references and causes CORD to immediately inform the current context elements whose keys match the given query references. Additionally, CORD will watch for changes of context elements and will inform the subscriber about the new value whenever the key of the changed context element matches one of the given query references. Changes of the value will occur due to insert, update, and delete requests or due to the expiration of a time-stamped value (which causes the use of the next alternative value as the current value). This way, from the time of subscription, the subscriber will always know about the state of all context elements known to CORD (or made known at any time in the future) whose keys match the given query references.

The conditional version is based on Event-Condition-Action rules (ECA rules). The only events supported are again changes of values in the sense of the previous paragraph; hence, the "event" part of an ECA rule is nothing other than a bag of query references wildcarding those context elements whose change of states should trigger the evaluation of the condition-action part. This latter part of an ECA rule is quite similar to the rules outlined in the previous subsection. That is, there are variables, and conditions are structured exactly the same way as in the case of conditional values, i.e. containers of comparisons of context and constant values in a switch-case construct. Unlike those rules, instead of a value, a sequence of actions can be specified for each case. Actions are operations that can be done by CORD, which include:

- Sending the current values of some specified context elements to the subscriber or other receivers.
- Sending a given literal message to the subscriber or other receivers.
- Inserting new context elements or updating existing ones; this triggers other events and may lead to indirect notifications.

Usually, the subscribers specify the bag of query references in ECA rules in such a way that the keys of all context elements that play a role in the condition part would match those query references. Such rules are quasi "alive": as soon as a case applies, it will be recognized. Therefore, they are the cornerstones for situation recognition for our context management service [5].

2.6 Insert, Update, and Delete

Insert and update requests must be submitted with the corresponding RDF descriptions, such as the one given in Sample 1. Delete requests, however, must contain a query description, which will lead to the deletion of matched context elements.

3 Summary and Future Work

We showed that contextual knowledge plays an important role in the Semantic Web and concluded that context management is *the* missing service in the Semantic Web. Our work contributes to filling this gap through the development of CORD, the context-oriented RDF database. CORD provides a solution based on the Semantic Web technology mainly for managing profiles. The concrete contributions of this paper are the introduction of: 1) a storage system for RDF-based profile data handling embedded resources, 2) a query language suitable for querying data organized in profiles, 3) a concept for storing rule-based values in profiles, and 4) a model for subscribing to context management services via the so-called event-condition-action rules.

We will continue this work by: 1) consolidating data from sources other than instances of CORD to satisfy the requirement #2 from section 1.1 completely, 2) equipping CORD with a special logic for reasoning about user location when sensory data is missing, based on information provided by PIM applications and the history of the location data, and 3) enhancing the existing privacy protection mechanism[13] by employing P3P and APPEL concepts when dealing with public service providers.

Acknowledgement. This work is partially sponsored by the Information Society DG of the European Commission. It is part of the MUMMY project (IST-2001-37365, Mobile Knowledge Management – using multimedia-rich portals for context-aware information processing with pocket-sized computers in Facility Management and at Construction Site) funded by the Information Society Technologies (IST) Programme. See http://mummy.intranet.gr.

References

1. Barstow, A (2001). Survey of RDF/Triple Data Stores. World Wide Web Consortium. Retrieved April 10, 2003 from http://www.w3.org/2001/05/rdf-ds/DataStore (last update Feb. 26, 2003).
2. Berners-Lee, T. & Hendler, J. & Lassila, O. (2001). The Semantic Web. Scientific American, May 17, 2001. Retrieved February 26, 2003 from http://www.sciam.com/print_version.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21.

[13] Currently, CORD provides access control mechanisms for the communication with context-providing and -consuming components that belong either to the user or to the communities of which the user is a member.

3. deVos, A. (2002). An RDF Query Language Based on DAML. Langdale Consultant, revision 1.0, Feb. 2002. Retrieved April 10, 2003 from http://www.langdale.com.au/RDF/DAML-Query.html.
4. Fikes, R. & Hayes, P. & Horrocks, I. (2002). DAML Query Language (DQL) – Abstract Specification. The Joint United States / European Union ad hoc Agent Markup Language Committee, August 2002. Retrieved April 10, 2003 from http://www.daml.org/2002/08/dql/dql.
5. Grimm, M. & Tazari, M.R. & Balfanz, D. (2002). Towards a Framework for Mobile Knowledge Management. Proceedings of the 4th international conference on Practical Aspects of Knowledge Management (PAKM2002), Vienna, Austria, December 2002.
6. Indulska, J. & Robinson, R. & Rakotonirainy, A. & Henricksen, K. (2002). Experiences in Using CC/PP in Context-Aware Systems. Proceedings of the 4th International Conference on Mobile Data Management (MDM2003), Melbourne, Australia, January 2003. Lecture Notes in Computer Science. Springer Verlag, LNCS 2574. pp. 247-261.
7. Karvounarakis, G. & Alexaki, S. & Christophides, V. & Plexousakis, D. & Scholl, M. (2002). RQL: A Declarative Query Language for RDF. ACM 1-58116-449-5/02/0005, WWW2002, Honolulu, USA, May 2002.
8. Klyne, G. & Reynolds, F. & Woodrow, C. & Ohto, H. & Hjelm, J. & Butler, M.H. & Tran, L. (2003). Composite Capability / Preference Profiles (CC/PP): Structure and Vocabularies. http://www.w3.org/TR/CCPP-struct-vocab/, W3C Working Draft March 25, 2003.
9. Kokkelin, S. (2001). Transforming RDF with RDFPath. Working draft, March 2001. Retrieved April 10, 2003 from http://zoe.mathematik.uni-osnabrueck.de/QAT/Transform/RDFTransform.pdf.
10. Magkanaraki, A. & Karvounarakis, G. & Anh, T.T. & Christophides, V. & Plexousakis, D. (2002). Ontology Storage and Querying, Technical Report No. 308. Foundation for Research and Technology Hellas, Institute of Computer Science, Information Systems Laboratory. Crete, Greece, April 2002. Retrieved April 10, 2003 from ftp://ftp.ics.forth.gr/tech-reports/2002/2002.TR308.Ontology_Storage_and_Querying.pdf.gz.
11. Melnik, S (2001). Storing RDF in a Relational Database. Retrieved April 10, 2003 from http://www-db.stanford.edu/~melnik/rdf/db.html (last update Dec. 3, 2001).
12. Miller, L. & Seaborne, A. & Reggiori, A. (2002). Three Implementations of SquishQL, a Simple RDF Query Language. Proceedings of the 1st International Semantic Web Conference (ISWC2002), Sardinia, Italy, June 2002. Retrieved April 10, 2003 from http://www.hpl.hp.com/techreports/2002/HPL-2002-110.pdf.
13. Schirmer, J. & Bach, H. (2000): Context-Management within an Agent-based Approach for Service Assistance in the Domain of Consumer Electronics. In: Proceedings of Intelligent Interactive Assistance, Mobile Multimedia Computing, Rostock, Germany, November 2000.
14. Sintek, M. & Decker, S. (2002). TRIPLE — A Query, Inference, and Transformation Language for the Semantic Web. Proceedings of the 1st International Semantic Web Conference (ISWC2002), Sardinia, Italy, June 2002. Retrieved April 10, 2003 from http://triple.semanticweb.org/iswc2002/TripleReport.pdf.
15. Tazari, M.R. & Grimm, M. & Finke, M. (2003). Modelling User Context. Proceedings of the 10th International Conference on Human-Computer Interaction (HCII2003), Crete (Greece), June 2003.
16. Tazari, M.R. & Plößer, K. (2003). User-Centric Service Brokerage in a Personal Multi-Agent Environment. To be presented in the International Conference on Integration of Knowledge Intensive Multi-Agent Systems (IEEE KIMAS'03), Cambridge MA (USA), October 2003.