A Context Management Infrastructure with Language Integration Support

Tobias Rho[1] Malte Appeltauer[2] Stephan Lerche[1]
Armin B. Cremers[1] Robert Hirschfeld[2]

[1]Department of Computer Science III
University of Bonn, Germany
{rho,lerche,abc}@cs.uni-bonn.de

[2]Software Architecture Group, Hasso-Plattner-Institute
University of Potsdam, Germany
{malte.appeltauer, robert.hirschfeld}@hpi.uni-potsdam.de

Abstract

A range of context-management systems in the past have motivated the need for development support of context-aware applications. They typically provide APIs and query languages for context analysis. Reacting to context changes, however, is either not at all or only to a limited extend supported by adhering to constraints of a framework.

In this paper, we present a context-management system that combines context reasoning with context-dependent behavior by taking advantage of language approaches to dynamic adaptation, such as aspect- and context-oriented programming. Our framework is open for different levels of integration with programming language extensions and offers a dynamic, strategy-based aggregation of local and distributed context sources. As a first step, we implemented a query library for the JCop language. We present its API and show the implementation of an example application.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Frameworks

General Terms Infrastucture, Context-Management

Keywords Context-Awareness, Semantic Web, Query Language, Context-oriented Programming

1. Introduction

Context-awareness is essential for a large number of today's applications. To build flexible applications that adapt their behavior to environmental changes, the underlying architecture must provide both the means to dynamically reconfigure the application based on new context information as well as stability of the applications against dynamic changes of context source configurations . Context-dependent reconfiguration consists of two important features, the ability to query many heterogeneous context sources – such as (Web) services, location sensors, contact list data, and others – as well as actuators of a dynamic adaptation – such as switching between different sorting algorithms. The former is addressed by context-management systems (CMS), which provide libraries and frameworks for context reasoning. These systems are often based on service-oriented architectures (SOA) and compose and configure services at runtime according to the current context.

The latter is also the focus of ongoing language design research mainly in the domain of context-oriented [6] and aspect-oriented programming [7]. The advantage of language-level approaches over CMS is that they do not impose restrictions to the design and implementation, such as subordinating an application to CMS-specific frameworks. However, they lack CMS-like context reasoning abilities. To make use of the expressiveness of these language approaches, it is desirable to have an integration of context-reasoning at programming language level equivalent to CMS capabilities. Tight language integration and static typing can also have drawbacks concerning runtime flexibility, therefore we present three different levels of language integration and their intended usage scenarios.

In a previous work [11], we presented an initial concept for the integration of a context-aware aspect language and an OSGi-based context management system which adapts to an architecture on the service-level. The approach used untyped Prolog predicates to represent context information. In this paper, we introduce our RDF-based context-management system, its logic-based object-oriented query language and its integration with a library for the context-oriented JCop [1] language. The intention of our context-management system is to provide a flexible context query and aggregation framework enabling tight programming language integration. The chosen context model is based on the *Resource Description Framework* (RDF), which represents contexts and their meta-data as object-graphs. RDF follows a *property-centric approach* which allows for the *extension of existing resource descriptions* rather than redefining them. Properties of a class can be defined in several parts that are aggregated to form a class specification. This allows for flexible aggregation of different context sources resulting in a combined, object-oriented model.

Section 2 presents a scenario of a context-aware mobile application to which we refer in the following sections. Section 3 presents our CMS, whose context query language is described in Section 4. An overview of the JCop context query library is given in Section 5. Section 6 discusses related work while Section 7 concludes the paper.

2. Running Example

Many mobile applications provide flexible behavior depending on context information. Consider, Zoe has a *'Getting Things Done'*-like ToDo application on her mobile device to help her prioritizing tasks depending on the current working environment and situation. For example, specific tasks require a computer, Internet access or can only be accomplished at a specific location. Others need

COP'11, July 25, 2011, Lancaster, UK.

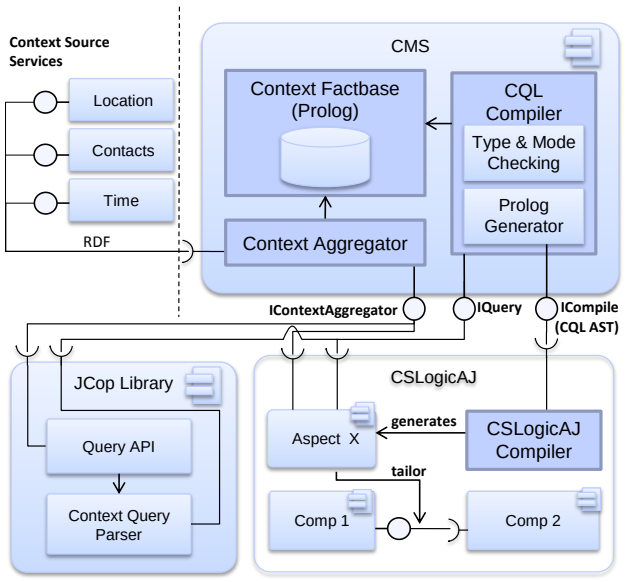

Figure 1. Overview of the Context Management System and the Query Framework

personal communication with a colleague or are due at a certain date. Zoe's application contains two entries:

- *"Arrange time and place for brunch on Sunday"*
  due: *Friday*, contacts: *<Mum, Dad, Uncle Malte>*
- *"Buy milk, eggs, juice"*
  location: *<supermarket at the corner>*

For the first task, the ToDo application checks whether one of the relevant persons is nearby (for personal communication) or online (for instant messaging). If so, it notifies Zoe of it at the startup of her calender application. For the second task, it rings when Zoe is close to the supermarket. One can think of many enhancements to the context reasoning and the according action in such application. For instance, instead of only notifying Zoe of the presence of contact persons, it could also, with regards to Zoe's schedule, send a request for a short meeting to them.

In our context query system, context information required by Zoe's application is gathered by several context sources, such as the device's GPS sensor, Web services and contact data. Each source offers an RDF context description that is gathered, optionally aggregated with similar contexts and transformed into a Prolog factbase. The RDF representation and processing of context data is described in Section 3. Context reasoning is expressed in our OCL-inspired logic query language that is introduced in Section 4. Context queries are first compiled, then transformed into Prolog queries and predicates and finally applied to the factbase. The query returns a result set, indicating if the required context is available, and provides a map of the query's variable bindings.

3. Context Management System

The context management system (CMS) is devided into three parts. A set of *context sources* provide context information and meta data. The context information is retrieved from sensors, local applications or external services. The context information is aggregated in a central CMS component. To use the CMS, a Client first requests a set of context sources to be queried - either by directly naming the source or by specifying properties that should be fulfilled by the context sources. Afterwards a client can evaluate queries on the CMS or register context listeners (called asynchronous queries), which are notified once the result of the corresponding query changes.

The CMS is based on the OSGi component framework. OSGi components, called *bundles*, are loosely coupled. They communicate via services registered in a service registry. Figure 1 illustrates the context management system and its intended usage. The core component of the framework is the CMS bundle. It contains a prolog-based *context factbase* and a *context aggregator* to feed the factbase from a set of *context source* services with RDF data modeled in RDFS. The context aggregator offers an API for requesting context sources via the `RequiredContext` interface, which is further elaborated in Section 3.2. The prolog engine is based on SWI-Prolog and its semantic-web library [13]. It provides a parsing framework for RDF, an RDF triple store with optimized hash indexes as well as predicates to evaluate the RDFS entailment.

Further the CMS contains the *context query language (CQL) compiler* which generates Prolog goals and predicates from CQL expressions. It can either be used statically for predefined queries or at runtime by passing a query string to the compiler. For static compilation the `Compile` interface to the CQL compiler receives a textual abstract syntax tree (AST) representation of the CQL query. The AST is type checked and translated to Prolog predicates and queries for which the framework provides a compiler, including a type checker and a Prolog code generator. At runtime the predicates are loaded into the context-management system and the queries are executed from byte code via the *IQuery* interface. The *IContextAggregator* interface enables a client to describe the necessary context information for the enclosing bundle.

Section 5 describes the embedding of the query language into a library for JCop where query strings are passed to the CMS at runtime. To take the full advantage of the type checker, an aspect or context-oriented programming language extension can embed the CQL grammar. This is realized in the aspect-language CSLogicAJ, which uses the CQL compiler to generate context-aware aspects statically.

3.1 RDF-based Context Representation

This section covers the mapping of RDFS class specifications to Java interface types that form the basis of the context-query language. The mapping is used for two purposes. First it specifies the Java objects returned by CQL queries and secondly it is the basis for the type system of the CQL. Since the description of the type system goes beyond the scope of this paper it will not be elaborated any further here.

The specification of an RDFS class can be distributed among several RDFS files, of which only a subset may be imported by any one client. To be able to statically type check our classes we only check against the RDF schemas imported by the aspect in question. The described types are therefore not globally defined, but only valid for the imported schemas in the current scope. For lack of space we only illustrate the general idea of the mapping.

Class and Property Mapping

RDF classes are mapped to Java interfaces. All properties and sub-properties[1] defined by `rdfs:domain` definitions are mapped to getter methods with an array return type, where the component type is the declared `rdfs:range` [2]. The common super interface for all classes is `URIReference`. It contains all properties whose domain is `rdfs:Resource`, `rdfs:Class` or do not have any domain declaration at all. The predefined `rdf:property Source` links each class and

[1] RDF models sub-properties via the *rdfs:subPropertyOf* relationship.

[2] The case of several range definitions goes beyond the scope of this paper.

```
ex:Contact rdf:type rdfs:Class.
ex:name rdf:type rdf:Property.
ex:name rdfs:domain ex:Contact.
ex:name rdfs:range xsd:string.
```

→

```
interface Contact
 extends URIReference {
  String[] name();
}
```

Figure 2. Exemplary mapping of the rdfs class Contact

property to a class `Source`, offering optional meta-data about the providing context source:

```
@prefix c: <http://www.iai.uni-bonn.de/context>.
c:Source rdf:type rdfs:Class.
c:Source rdf:type rdf:Property;
       rdfs:domain rdfs:Resource;
      rdfs:range c:Source.
```

By default it does not contain any properties.

Context sources, which will be discussed further in Section 3.2, may additionally define arbitrary meta data, e.g. accuracy, in their RDF Schema definition.

Figure 2 illustrates the RDFS mapping with the class `Contact` and one property `name` with range `xsd:string`. All primitive data types defined in XML Schema [8, 3.2] are represented as Java basic types. RDF containers and collections are represented in the same way, but marked as collection properties[3]. Since `rdfs:range` definitions do not allow for parametric polymorphism the return type of the method is always `Object`[4].

We adopted the prefix namespace binding from XML namespaces [8] in order to separate the simple name from a URI and also to distinguish classes and property names in different namespaces. In contrast to XML namespaces the prefixes on property and class names only need to be specified if the simple name is ambiguous.

Depending on the level of language integration there are three options of using this mapping from a concrete language or client:

1. The types are represented as a generic graph with nodes of type IContext defined in Figure 3. The edges are traversable via `getObjectsForProperty`. The JCop example in Section 5 use this approach.
2. The code of a language is statically compiled against the interfaces in a certain context, e.g. a context-aware aspect or layer refers statically to a set of imported RDFS types. In this variant the Java types do not exist at runtime and internally the generic graph of `IContext` objects is used for the implementation.
3. Java interfaces are generated from the RDF mapping and used in a Java client that accesses the CMS.

The first approach is the most flexible, but does not offer static compilation and processing queried context objects takes a lot of effort since object properties are represented in a generic fashion by key value pairs and subtype relationships are not reflected by a Java hierarchy. The second approach is suitable for Java language extensions which incorporate the RDF Java mapping, without realizing are concrete Java class generation. This makes it possible to statically compile e.g. an aspect A_1, against a set of imported RDF schemas and another layer A_2 against another set of schemas, resulting in two different sets of RDF classes.

The third variant is suitable when query results are referenced from regular Java code and a common set of RDF schemas is used throughout the whole application. In this case a set of Java classes is statically generated and can be referenced by code compiled by a common Java compiler. This step is comparable to other static RDF Java generators[5].

[3] They will be handled differently by the context query language.

[4] Assuming the namespace definition: `namespace rdfs = "<http://www.w3.org/2000/01/rdf-schema#>";`

```
public interface IContext {
  public Object[] getObjectsForProperty(String fieldName);
  public String getContextClass();
  ...
}
interface IContextSource {
  String getId();
  String[] getRDFSchemas();
  void start(IContextAggregator aggregator);
  void stop();
  InputStream getSnapshot();
}
```

Figure 3. Generic context classes `IContext` and `IContextSource`

```
interface NearbyService extends IContextSource {
  @RDFSClass(type="http://example.com/Nearby",list=true)
  ISnapshot nearby(Map<String,Object> parameters);
}
interface ISnapshot {
  int getExpirationTime();
  InputStream getData();
}
```

Figure 4. IContextProvider

3.2 Local and Query Context Sources

Local Context Sources are represented as OSGi services. The complete context data of a local context source is stored in the Prolog database. They implement the interface `IContextSource`, see Figure 3. Context sources may compete with each other, e.g. different location sensors provide the same context classes but may use different means to acquire them, like GSM triangulation and GPS. They can provide dynamic meta information about themselves, like precision, in key value pairs as OSGi service properties. Section 3.3 will explain how this meta data can be used in service requests.

There are two options to configure local context sources. Either they push data incrementally to the context aggregator, which is started by the `start()` method, or the context aggregator pulls snapshots of a sensor, e.g. the current gps position via `getSnapshot()`.

A range of approaches (e.g., [10]) extend OSGi to a remote framework. Based on these frameworks, context sources can be remote OSGi services located on a different node. Still, they feed the prolog database with the complete data-set of a context source. But for some contexts, access to only a subset of the source's data is necessary. For example, if it refers to a large database, it is desirable to consider only specific data, not the whole content. Examples for such context sources are social network APIs that provide access to their databases via Web services. In this case, *Query context sources* can be applied. *Query context sources* are implemented in pure Java and therefore we cannot refer to mapped RDF classes. They implement the `IContextSource` interface, as does any other context source, and additionally contain a method for each query to the external service. Each method has an `@RDFSClass` annotation attached, representing the RDF class returned by the method.

The only parameter that is passed to the method must be of the type `Map<String,Object>`. It takes key value pairs with values of type string or basic types. The parameter map is kept generic, to easily integrate RESTful webservice interfaces with non-fixed argument sets. Figure 4 illustrates this with the nearby service interface which is wrapped in an equally named method that can be linked to a localization service such as Google Latitude for instance. This context source will later be used in a JCop layer to

[5] e.g. RDFReactor (http://semanticweb.org/wiki/RDFReactor), Sommer (https://sommer.dev.java.net/)

retrieve nearby contacts. The `nearby` method expects the parameter `maxDistance`, providing the maximum distance in meters to nearby contacts. The type declaration in the `@RDFSClass` annotation describes the returned RDF type of the method which is necessary for the static type safety of the query language. The connection to the service and the mapping to RDF falls into the responsibility of the `IContextSource` itself. The queries on external context sources are executed via a Prolog Java Bridge, which is part of SWI-Prolog's bidirectional Prolog/Java interface JPL. Since accessing remote data is a time consuming operation, caching is an integral part of our framework. All data belonging to the same query and containing the same input parameters is cached until the expiration time is reached. The returned triples are added to the factbase and can be queried by later expressions, following logical update semantics [3].

3.3 Requesting Context

The framework offers two different means for requesting contexts. A context requester may either request a concrete context source, by referring to their services, or to context classes, by referring to RDF schemas containing the required context classes.

The former is necessary if a specific sensor, e.g. a GPS sensor, is needed or different sensors providing the same context information should be aggregated. The latter is useful if only the context kind, e.g. location information, is relevant but the actual context source may be chosen or switched automatically by the system. Additionally, both requests can be combined with an *LDAP* (Lightweight Directory Access Protocol)[2] filter string to filter context sources via meta-data attached to the context source service[6].

The LDAP was originally developed as a directory service to manage information about users, hardware and services in a company network. The LDAP standard defines a filter language to search for (filter) resources based on attributes attached to the resource. Each attribute stores a single type of information. The predefined attribute *objectclass* represents all class (interface) names under which a service was registered to the system. OSGi services are optionally registered with a set of attributes attached to the service, e.g.

```
public void start(BundleContext bc) {
  Hashtable props = new Hashtable();
  props.put("quality", "10");
  LocationSensor sensor = ...;
  bc.registerService(LocationSensor.class.getName(),
   sensor, props);
```

LDAP filters include less-than, equal and greater-than checks on the service attributes which can be combined by logical operators. The attributes can be updated at runtime by altering the properties on the service reference.

In case the properties change, a service event is fired and all registered listeners are notified. The properties are added to a context source as described in Section 3.2.

The filter allows for fine-grained selection of context-sources based on attributes attached to the context sources. The predefined filters include equality and comparison checks of attribute values of a single source. But they lack the comparison of attributes across different services. Therefore we added the `min(attribute)` and `max(attribute)` conditions which are true for the sensor with the maximum property value. If several services have the same attribute value the condition is true for an arbitrary service. Figure 5 shows the `IContextAggregator` interface, which Clients use to requested context sources. The members of the annotation have the following meaning:

[6] LDAP filter matching of services is already built into OSGi service queries.

```
public interface IContextAggregator {
  public IContextRequest requestSource(Bundle bundle,
     Class source, String strategy, Cardinality cardinality);
  public IContextRequest requestSchema(Bundle bundle,
    String schema, String strategy, Cardinality cardinality);
  public void removeRequest(IContextRequest schema);
}
public enum Cardinality { single, multiple }
public interface IContextRequest {
  boolean isAvailable();
  void dispose();
  ...
}
```

Figure 5. Requesting Context - Dynamic Approach

source A context source class registered as a service.

schema The requested schema definitions.

strategy A filter expression following the syntax described above, which refers to the corresponding source or schema. If left empty no further strategy is applied. The system will use the filter (objectclass=org.cs3.ditrios.context.IContextSource) in this case.

cardinality A member of the Cardinality enumeration listed in Figure 5 with the options single and multiple source. In case of the *single* option only one (arbitrary) context is chosen which matches the filter and is the default setting. Otherwise all matching context sources are selected.

The contexts can also be requested and removed dynamically via the `IContextAggregator` interface, see Figure 5. The `IContextAggregator` API allows the clients to change the requested contexts at runtime based on either program state or user interaction. An example for the latter might be asking the user which address book service should be used to reason about contacts in a context-sensitive application. In case only an RDF schema is requested the context source might change transparently. Lets assume a client requests a Location schema with maximum precision. While indoor a WLAN localization context source is used. Once the device is moved outside and a good GPS signal can be acquired the CMS switches to GPS localization transparently for the client. Each request is represented as an `IContextRequest` instance and must be removed, once the context is not needed anymore. This happens automatically once the enclosing OSGi bundle is stopped.

4. Context Query Language

The Context-Query Language (CQL) is based on the syntax of the Object-Constraint Language (OCL). We have chosen it as a role model because it is well known to developers, provides good means for constraints on object graphs and has a simple syntax for object graph traversals. We further integrated logic variables to query for context objects. We only sketch the language in this paper, since we focus on the context-management system.

Contexts are queried by predicates over the Java RDF types, as described in the previous section. We can refer to the set of instances of a class with its class name C. The language provides a number of operations on context sets, aligned to the OCL names: *select*, *reject, forAll*, *exists* and additionally the *one* operation. The following expression selects all context instances bound by the expression e which fulfill the condition c:

e->select(c)

Properties on a list of contexts are mapped to each list member individually, meaning for all list elements for which the property exists, the property is added to the returned list. So the following expression matches all properties p of instances of C fulfilling c:

C->select(c).p

The *one* operation is special to the CQL compared to OCL. It binds one context instance at a time and the CQL backtracks over all elements of a list:

e->one(c)

The condition c is evaluated in the context of the queried type. If the property has an `rdfs:range` of rdf container or collection v will be bound to the array of all elements, otherwise it will backtrack over all defined properties, e.g. in case `Contact->one().name` over all defined `name` properties. The instance of the queried class may be named; in the following example, the current contact instance is bound to `c`.

Contact-
>one(c | c.firstName=$value_1$ && c.lastName=$value_2$) {...}

All logic variables must be declared at the beginning of the expression enclosed in parenthesis and separated by a colon from the query. Variables and expression can be unified via the unification operator "=", e.g.

(Contact c): c = Contact->one()

binds the variable `c` to a `Contact` object and the CQL backtracks over all contact object in the context factbase.

Accessing query context sources In Section 3.2 we introduced context query sources, they take a set of parameters and return a structured set of data. In the CQL they are queried by the following expression:

var = SensorTypeName->methodName({$param_1 = value_1$,...})

Continuing the nearby contacts example, a query returning informations about contacts within a maximum distance of 100 meters can be written as:

```
(Nearby[] nc):
  nc = NearbyService->nearby({maxDistance=100})
```

Universally quantified statements over context lists are written as e->forAll(c) and the existential quantification is written as e.exists(c). The language supports all primary logic operations (and, or, not), e.g. the following conditions ascertains that all contacts from the given list `CS` are nearby and that their distance to the current position is less than the value `max`.

```
(Contact[] cs, int max) : cs->forAll(c | NearbyService->
      nearby({maxDistance=max})->exists(c.email=email))
```

5. JCop Query Library

We developed a library for JCop [1] as an interface to the query system. JCop is a language extension to Java implementing the context-oriented programming approach [6]. It provides first-class layers (modules that encapsulate variations of methods) and explicit, implicit, and declarative constructs to control layer activation at run time (controlling which method variation a call is to be dispatched to). Offering these constructs, JCop supports means to modularize and control context-depended functionality. Context reasoning, however, is not explicitly supported, yet.

5.1 Overview

Our JCop query library supports executing context queries and defining actions - for example, layer activation - to be taken on context change. In the following, we briefly describe the most important API objects and methods[7].

`ContextQuery(ContextRequest, String, String)` A query object is instantiated with its context type schema, a default namespace, and a string representation of the CQL expression.

`boolean ContextQuery.evaluate()` A query can be executed synchronously and will immediately return a Boolean value whether the context is constituted or not.

`void ContextQuery.evaluate(IContextHandler)` Queries can also be executed asynchronously. In that case, an `IContextHandler` object is passed to the query's evaluation method that is called by the CMS on context entry and exit.

`ResultSet ContextQuery.getResultSet()` The variable bindings of the last executed query are represented by a `ResultSet` that holds a list (for each solution found via backtracking) containing maps of key-value pairs. In addition, `ResultSet` provides some auxiliary methods such as `boolean isEmpty()`.

`void ContextQuery.addLayers(Layer[])` Layers can be associated with a query, for example, to make them accessible to the `IContextHandler` callback methods.

`void IContextHandler.onContextEntry / Exit(Layer[])` The callback methods are activated on context change and can be used for implementing any kind of reaction to the new state. They are parameterized with `Layer` objects if they have been associated with the query.

5.2 Example

In the following, we sketch a decomposition of the ToDo application example using our context query library. The example in Figure 6 presents three main activities: *modularized definition of adaptation code* with layers, *context reasoning* with the query language, and *activation of the adaptation* by a context object .

The layer `NearbyContactsMsg` implements the nearby contacts message display on interaction with the calendar application (Lines 6–15). The layer and its partial methods require context data in form of references to the nearby contact objects. This information is passed to the arguments of the layer constructor[8]. Layer activation is controlled by a `on` predicate encapsulated in the `NearbyContacts` context object. The predicate is evaluated on every execution of a layered method; if its boolean expression returns true, it activates the nearby contact layer for this method execution. The context object is created and activated during generation of the corresponding `ToDoItem` (Lines 20–21). The context query is created on instantiation of `NearbyContacts`. First, we request a context schema from the `ContextAggregator` that defines the context types to be used in our query (Lines 35–40) and declare the default RDF namespace (Line 43). With that, we are able to specify the actual query as a string, using `Contact` and `NearbyService` and their properties. The query first selects all `Contact` entities that fulfill the condition of the `select` predicate, i.e. where first and last name match a `Contact` object (associated with the `ToDoItem`). These entries are then filtered by a `forAll` predicate. It selects only those entries for which the `NearbyService` context type finds a match within a range of 100 meters (Lines 44–47). We use JCop's `on` predicate to evaluate the context query at any *layered method*, that is, any method potentially affected by the respective layer activation (Lines 28–29). In this example we use the synchronous query evaluation that returns a Boolean value indicating that the context is active.

6. Related Work

Most context-aware platforms concentrate on the context management providing a well-defined interface for an application. The ap-

[7] The classes `ContextQuery` and `IContextHandler` belong to the library package `jcop.query`; `Layer` refers to `jcop.lang.Layer`.

[8] Layer and context instantiation has been recently introduced as a new feature to the JCop language.

```
public class CalendarApp {
  public void initialize() {...}
  ...
}

public layer NearbyContactsMsg {
  private void ResultSet contacts;

  public NearbyContactsMsg(ResultSet rs) {
    thislayer.contacts = rs;
  }
  before public void CalendarApp.initialize() {
    //show message which contacts are nearby
  }
}

public class ToDoApp {
  public addContactsToItem(Contact[] cts) {
    ... //create a new ToDoItem
    ctx = new NearbyContacts(cts);
    ctx.activate();
  }
}

public context NearbyContacts {
  private ContextQuery nearby;

  on(nearby.evaluate()) :
   with(new NearbyContactsMsg(nearby.getResultSet()));

  public NearbyContacts(Contact[] cts) {
    this.nearby = createQuery(cts);
  }
  private ContextQuery createQuery(Contact[] cts) {
    ContextRequest request =
      CMS.getContextAggregator().requestSchema(
        null,
        "http://www.example.org/nearby.rdf",
        "max(precision)",
        Cardinality.single);
    return new ContextQuery(
      request,
      "http://www.example.com",
      "Contact->select(" + createCond(cts) + ")
         ->forAll(c | NearbyService
           ->nearby({maxDistance=100})
             ->exists(c.email=email))");
  }
  private String createCond(Contact[] cts) {
   // for each contact c in this.cts, generate:
   // ( firstName=c1.getFirstName() &&
   //   lastName=c1.getLastName() ) || ...
  }
}
```

Figure 6. Implementation of the ToDo application using JCop's query library.

plication anticipates the interfaces to the context management and is therefore itself context-aware. One example is the SOCAM approach [5] by Gu et al., an approach built on top of the OSGi framework. Context reasoning is carried out with a rule system based on first-order logic. The context model is described with the help of the Web Ontology Language OWL [9]. Automatic adaptation of the program based on the current context state is not supported.

Semantic Space [12] focuses on three main tasks: to provide an explicit representation of the raw context data, to provide the means to acquire contexts via expressive queries and thirdly to provide high-level contexts through reasoning. The context itself is modeled as an ontology, *context wrappers* provide these contexts, they are discovered and handled by a context aggregator. Four basic classes can be used to characterize smart spaces; user, location, and computing entity as real world objects as well as the activity as a conceptual object. Though conceptually very similar, it does not offer tight language integration, no mapping of contexts to Java objects and the query language is not as expressive as our CQL.

Du and Wang developed a programming approach for context-awareness on mobile devices [4] that supports method interception at each state transition, that is, at each entrance and exit of contexts. The context itself is represented by a set of sensor values. Reacting to context changes is restricted to the use of callbacks. Context sensor values are restricted to numbers and are polled at an interval rather than by the application on demand.

7. Summary and Future Work

This paper presented a context-management framework which allows for tight integration with object-oriented languages. We focused in this presentation on a context model representation and infrastructure for requesting and querying context sources and how they can be integrated into the context-oriented programming language JCop. The properties of the context query language were only outlined in general. Future work will introduce the statically typed context query language and how it can be integrated into the aspect-language CSLogicAJ [11]. For more-complex queries it offers the definition of recursive predicates, which are reusable by utilizing a generic type system. Furthermore the access of context source meta-data will be illustrated.

References

[1] M. Appeltauer, R. Hirschfeld, H. Masuhara, M. Haupt, and K. Kawauchi. Event-based Software Composition in Context-oriented Programming. In *Proceedings of the 9th International Conference on Software Composition*, Lecture Notes in Computer Science, pages 50–65, Berlin, Heidelberg, Germany, 2010. Springer-Verlag.

[2] B. Arkills. *LDAP Directories Explained: An Introduction and Analysis*. Addison-Wesley Professional, 2003.

[3] E. Boerger and B. Demoen. A framework to specify database update views for Prolog. *Lecture Notes in Computer Science*, 528, 1991.

[4] W. Du and L. Wang. Context-aware application programming for mobile devices. In *C3S2E*, volume 290 of *ACM International Conference Proceeding Series*, pages 215–227. ACM, 2008.

[5] T. Gu, H. K. Pung, and D. Q. Zhang. Toward an OSGi-based Infrastructure for Context-Aware Applications. In *IEEE Pervasive Computing, vol. 03, no. 4*, Oct-Dec 2004.

[6] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented Programming. *Journal of Object Technology*, 7(3):125–151, March-April 2008.

[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented Programming. In *Proceedings 11th European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, 1997.

[8] A. Malhotra and P. V. Biron. XML schema part 2: Datatypes second edition. W3C recommendation, W3C, October 2004. http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/.

[9] C. Welty M.K. Smith and D.L. McGuinness. Owl web ontology language reference. http://www.w3.org/TR/owl-ref, Feb. 2004. World Wide Web Consortium (W3C) recommendation.

[10] J. S. Rellermeyer, G. Alonso, and T. Roscoe. R-osgi: Distributed applications through software modularization. In *Middleware*, pages 1–20, 2007.

[11] T. Rho, M. Schmatz, and A. B. Cremers. Towards context-sensitive service aspects. In *In Proceedings of the Object Technology for Ambient Intelligence and Pervasive Computing Workshop, July 3-7, Nantes, France*. July 2006.

[12] Xiaohang Wang, Jin Song Dong, ChungYau Chin, SankaRavipriya Hettiarachchi, and Daqing Zhang. Semantic space: An infrastructure for smart spaces. *IEEE Pervasive Computing*, 3:32–39, 2004.

[13] J. Wielemaker, M. Hildebrand, and J. Ossenbruggen. Using Prolog as the fundament for applications on the semantic web. In *Proceedings of the 2nd Workshop on Applicatiions of Logic Programming and to the web, Semantic Web and Semantic Web Services*, volume 287 of *CEUR Workshop Proceedings*, pages 84–98. CEUR-WS.org, 2007.