Supporting Context-Aware Applications: Scenarios, Models and Architecture[*]

Ricardo Couto Antunes da Rocha[†], Markus Endler

[1]Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)
Rio de Janeiro – RJ – Brazil

{rcarocha,endler}@inf.puc-rio.br

Abstract. *Context-aware computing is widely accepted as a promising paradigm to enable seamless computing. Several efforts have been made in order to support context-aware applications through the development of software infrastructures, middlewares and models for describing context information. However, developing such applications is still a complex task because of the lack of adequate software abstractions, programming models, methodologies and efficient middleware. This paper presents an approach for context-aware software development based on a flexible context model and an infrastructure for evolutionary management of context information. We present a context model that provides high-level abstractions to manage and handle context information. In order to demonstrate the capability of the proposed approach, we present a case study of a location-aware instant messaging application.*

1. Introduction

Context-aware computing is widely accepted as a promising paradigm to implement seamless computing. Several research efforts have been made in order to enable such paradigm, producing a number of prototypes, middleware and frameworks. Context Toolkit [Dey et al. 2001] and RCSM [Yau et al. 2002] are examples of middlewares for context-aware computing. However, developing context-aware software is still a complex activity because of the absence of general and well-accepted software abstractions, programming models and methodologies, as well as efficient middleware. Currently, research in the field of context-aware systems addresses such needs by developing middlewares and suitable context models.

Middleware for context-aware systems still suffer from some limitations, such as lack of scalability and problems to support heterogeneous environments [Henricksen et al. 2005]. Recent proposals agree about the need to hide from the application developer some of the complexity of context processing, such as details concerning sensor management or context storage. However, yet there is not a consensus about what layers of abstractions are required for a context-aware middleware. Moreover, middlewares still lack suitable programming models for selecting context-based adaptation.

[*]Partially supported by CNPq research grants 55.2068/02-2 (ESSMA) and 479824/04-5 (Ed. Universal).
[†]Supported by CNPq

Research in context modeling has focused on the development of models capable of describing the complexity of context information, such as dependency among contexts, complex inference rules, inconsistency and ambiguity resolution, and description of privacy requirements. In this respect, a promising approach is the use of ontologies for describing context. However, research has neglected some issues such as the trade-off between flexibility of context models and its usage by resource limited devices. Another issue that demands attention is the integration between modeling concepts and programming abstractions, especially for describing context-specific abstractions such as spatial queries for location context.

These new requirements motivated us to extend the MoCA middleware [Sacramento et al. 2004] in order to implement a more flexible approach for managing context information, and to develop context-related programming abstractions which are better suited to common application needs.

This paper presents an approach for context-aware software development based on a flexible context model and an infrastructure for context management, targeting continuous evolution of context, heterogeneity of the devices, and exploring context-based programming abstractions. We discuss how context models and a middleware can be used in conjunction to specify context-specific abstractions that support a disciplined software development for context-aware systems. We present a context model that explores such opportunities and provides higher level abstractions to manage and handle context information. This context model offers constructors for describing quality of context, contextual events and context queries. We argue that the use of such constructors can significantly reduce the software's complexity and improve code reuse. In order to demonstrate the proposed approach, we present a case study of a location-aware instant messaging application.

The remainder of this paper is structured as follows. Section 2 discusses some requirements for developing context-aware systems, in terms of middleware support and context models. Section 3 briefly describes our infrastructure for deploying and managing context. Section 4 presents our proposal for a flexible context model. Section 5 presents a case study of the implementation of a location-aware application using the proposed approach. Finally, section 6 compares the proposed approach with related work, and section 7 presents the current state of our research and points to future research directions.

2. Towards a Support for Context-Aware Software Development

The development of context-aware system demands support for both context-aware, adaptive applications, and sensors that generate the actual context data. At sensor level, it is important to model the idiosyncrasies of sensor management, such as precision modeling and methods to resolve ambiguity of sensors readings.

These requirements impact the design of both middlewares and context models. Indeed, they are strongly interdependent since the complexity of a context model determines the complexity of context management by a middleware. Coutaz *et al* [Coutaz et al. 2005] presents this relationship as a conceptual framework that interconnects an ontological foundation for context modeling with a runtime infrastructure (middleware). We believe that such interdependence has been neglected in most middleware projects.

This paper does not intent to discuss all requirements of context-aware systems. For a more systematic discussion, please refer to [Raatikainen et al. 2002], [Strang and Linnhoff-Popien 2004], and [Henricksen et al. 2005]. Conversely, our work focuses three main requirements: support in heterogeneous environments, management of context evolution, and support for context-specific abstractions. The heterogeneity and evolution requirements have been already discussed in an earlier paper [da Rocha and Endler 2005].

A context model should provide an unambiguous definition of the context's semantics and usage. This information is both necessary for developing middleware that correctly processes and manages context information, and for communicating to developer of context-aware applications how context data can be used to trigger context-dependent adaptations. Therefore, we believe that context modeling is part of the software engineering process and that it impacts on several, if not all, phases of the software life cycle. However, few work (e.g. [Henricksen and Indulska 2006]) have explored this practical aspect of context modeling for context-aware software engineering. The support for context-specific programming abstractions is an example of aspects that have not been enough explored in most current context models.

In context-aware computing, the software developer must specify how applications and the infrastructure (e.g. middleware, services) should change their behavior when context changes. Currently adopted paradigms for programming adaptations often focus on general-purpose approaches such as synchronous or asynchronous communication, use of profiles for describing context-based adaptations [Chan and Chuang 2003] and computational reflection [Capra et al. 2003, Chan and Chuang 2003]. Since these paradigms are totally decoupled from the underlying context models, they do not give direct support for new abstractions that may be required for handling some context information. Indeed, the deployment of a new context type may introduce new opportunities and requirements for development of adaptation code supposed to handle the new environmental situations that the context describes. A new context type may also introduce new context-specific abstractions, such as queries and high-level conditions.

For example, consider modeling a device's location context in two ways: either as a pair of coordinates representing latitude and longitude, or as a symbolic region, such as *room A* or *building B*. The deployment of this new context type in a context-aware system may require new programming abstractions for location-aware applications, for example:

- Spatial-based queries for retrieving the set of devices that can be found in a certain physical area, or the Euclidean distance between the device and a reference point.
- New forms of specifying precision of a location. For example, an application may request location information at different granularities, e.g. single rooms, floors of a building, buildings, blocks or campi.
- Location-aware applications may need to react according to very specific location events, such as the change of a device's location, or its entry in (or exit of) a symbolic region.

MiddleWhere [Ranganathan et al. 2004] provides several examples of other interesting abstractions and algorithms for handling location context. However, all of them are restricted to the domain of the location context. Similarly to the case of location, also other context information may require the adoption of specific queries, events or conditions for firing adaptations.

We believe that such context-specific abstractions are imperative for the development of complex context-aware applications, but they are very difficult to support by generic means. In fact, only the designer of the context model can specify all abstractions introduced by new context information. Moreover, leaving the task of interpreting context-specific abstractions to the application developers has two main disadvantages. Firstly, developers might make wrong inferences about context information as they usually do not have full understanding of context internal semantic. The second disadvantage is that useful abstractions might not be shared between applications, thus hampering code reuse and increasing application complexity.

3. An Infrastructure for Context Management

In order to address the before mentioned challenges, we have designed a Context Service for the MoCA middleware. This Context Service targets flexibility, allowing runtime incorporation of new context information types and new context inference agents. We designed both the Context Service architecture and defined the context modeling approach, as they are closely related.

The design of the Context Service was driven by the following requirements: (1) adoption of a generic and flexible context model that could be dynamically deployed in the middleware architecture; (2) improved performance, allowing fast access to and dissemination of context information, avoiding the intensified use of network and memory in limited-resource devices; and, (3) support for context-related interoperability, facilitating the use of the service from different devices, operating systems and programming languages.

In our approach, essentially three components interact to create, disseminate and use context information: *context provider*s, *context consumer*s and the *Context Service*. These components are network entities, represented by applications, services or hosts. The context provider is an entity responsible for publishing certain context information; it can be the generator of a raw data or just an interface between a sensor and the middleware. The context consumer is an entity interested in a given context information. The Context Service is responsible for receiving, storing and disseminating context information.

The *Context Service* is composed of the basic elements shown in Figure 1. *Event Service* is responsible for providing asynchronous communication, i.e. delivering context information and contextual events to interested consumers. The Event Service adopts a publish/subscribe paradigm and offers a specialized API to handle subscriptions for contextual events. The *Type System Manager* maintains the dynamical context type system, solving and recognizing context types at runtime. The *Repository* maintains the database of context data.

One of the cornerstones of our approach is a strongly typed context model. This model is mapped to object-oriented language constructions which are incorporated to applications accessing context information. Application-specific context is modeled and handled using the same approach. We follow an OO model for context handling, instead

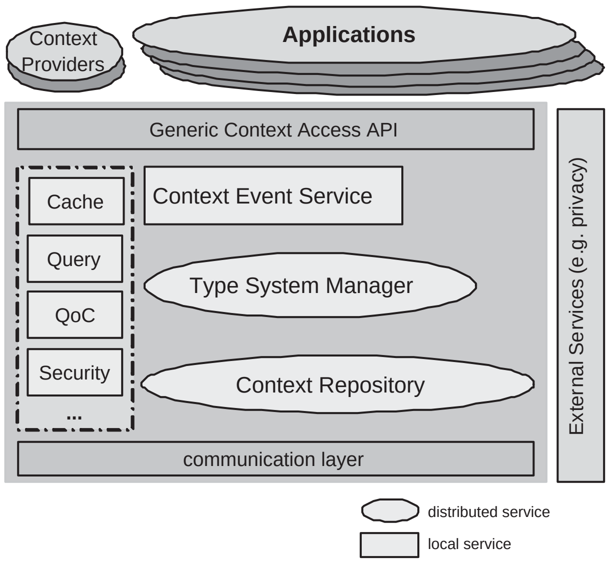

Figure 1. Context Service Architecture

of an ontology-based model, because the latter requires resource-hungry ontology engines which hinder context management on resource-limited devices.

The deployment of a new context type requires two main steps: context modeling and the context model processing. The first step consists of modeling the new context information using an XML-based approach. In an XML file, the context modeler specifies attributes, characteristics, relationships with previously specified context, quality-of-context attributes and the queries for synchronous context request. Section 4 describes our approach for context modeling.

In the context model processing step, a *Context Tool* reads the XML file and executes the following tasks: (1) Validates the XML syntax and the context model; (2) Updates the context type system and initializes the repository for storing the new context information; (3) Generates a library containing the language bindings for the access operations of the deployed context. So far, we have just implemented the Java language binding for context access.

When an application developer needs to use a context information, he/she reads the XML file with the context's model to understand the context semantics, and includes the binding library in his application. The language bindings allows the application to access context information in form of object references or attributes, and query contexts using object methods. Dependencies among context types are also included in the binding file. For additional information about the architecture of our Context Service, please refer to [da Rocha and Endler 2005] and [da Rocha and Endler 2006].

4. Context Modeling Approach

In the MoCA architecture, we have modeled two base-level context information: a *local context* information that describes the context of a device including data on energy level, memory and CPU usage, and *wireless connectivity* information about reachable IEEE 802.11 access points and the corresponding signal strengths. The MoCA *Monitor* is a daemon executing on each mobile device, and is responsible for publishing this base-level context information. *LIS* (Location Inference Service) [Nascimento 2005] is a MoCA service responsible for publishing the location context in terms of symbolic locations (e.g. *Room A*, *Building B*), which are inferred from the device's wireless connectivity context. Finally, SRM (Symbolic Region Manager) is a service responsible for maintaining information about the (hierarchical) structure of symbolic regions and for mapping them to areas in maps, previously registered on LIS.

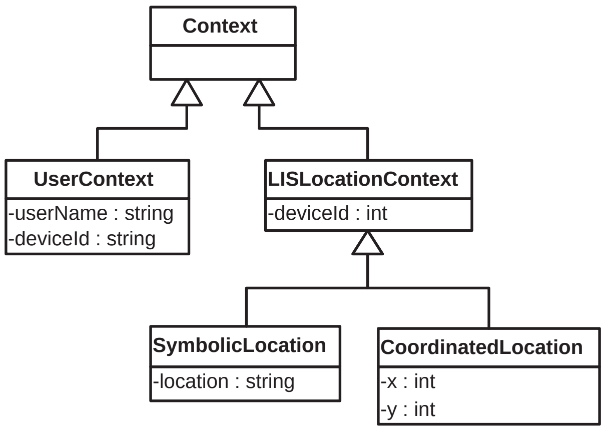

Figure 2. Example of context modeled for MoCA

Figure 2 shows an example of context modeled for MoCA architecture, including the location context provided by LIS. The following simplified XML shows the modeling for `UserContext`.

```
<context name="UserContext" base="Context" ... >
   <attribute name="userName" type="String" ...>
      descriptive comment
   </attribute>
   ....
   <queries>
      <query name="onlineUsers" resultType="UserContext">
         <param name="location" type="SymbolicLocation"/>
      </query>
      ....
   </queries>
   <events>
      <event name="EnteringInAnArea" ... >
         <condition type="ProviderDefined" />
      </event>
   </events>
</context>
```

Our context model is composed of four parts: *context structure* (sections 4.1 to 4.3), which represents the data attributes and the interrelationship among context types, *quality of context meta-information* (section 4.4), *contextual events* (section 4.5) and *queries* (section 4.6).

4.1. Attributes

Attributes are the basic units holding some information about a context, e.g. a numerical or symbolic value. For example, in the figure 2, `userName` is an attribute of `UserContext`.

4.2. Associations

Associations implement a containment relationship between two context types. They are used when an attribute is of certain context type. In terms of implementation, this relationship is simply a link from a context information to another, and allows the navigation through the context graph along these links.

Although our model is mapped to an object-oriented model, the associations do not embody the meaning of aggregation, hence each context information has its own life cycle. Associations also define a dependency relationship because a change in the associated context causes a change in the context that contains the association. Inference (i.e. when a context information is translated into another context information) and other possible dependency relationships are not implemented in our model as an explicit relationship.

4.3. Inheritance

Our context model also supports the implementation of the inheritance relationship between two contexts types. This relationship allows the construction of a context from another preexisting context. The new context type inherits attributes, associations, events and queries from the more generic type, and thus supports reuse of models.

Inheritance also implements the sub-typing relationship among contexts. The sub-typing relationship is the core concept supporting heterogeneity of context providers and the evolution of context [da Rocha and Endler 2005]. As shown in the example, any context is a sub-type of `Context`.

4.4. Quality of Context

Quality of context (QoC) is modeled as meta-information associated with context attributes. Our model allows the specification of two dimensions of QoC: *precision* and *accuracy*.

Context precision specifies a range of values associated with a context attribute value. For example, for a location information retrieved from a GPS sensor, the precision is usually a constant value, e.g. approximately five meters. In this case, the precision informs the presumed maximal error of the context data. Precision can also be specified in terms of a context-specific structure, instead of a numeric value. For example, when the LIS service delivers a location context information in terms of symbolic regions, the precision is also described in terms of symbolic values, such as `room`, `building` or `campus`, and thus the precision assumes the meaning of information granularity.

Accuracy specifies a numerical value to estimate how correct a context data is. For example, the accuracy of a symbolic location generated by the LIS service is very much dependent on the number of 802.11 access points that are used to infer the device's position. Our model also supports the definition of time-based accuracy, i.e. represented by the moment in which the context information has been acquired from the sensors. The following part of a XML model shows an example of QoC attribute declaration.

```
<context ... >
  <attribute name="GPSLocation" ... >
   <qoc>
```

```
      <precision type="int" static="yes" name="error"
                 value="5"  unit="meters">
      </precision>
      <accuracy ... />
   </qoc>
  </attribute>
  ...
</context>
```

There are other meta-information, such as freshness, but they are not handled as QoC parameters. The main difference between these and the aforementioned meta-information is that only QoC parameters can be used for *QoC negotiation*. Negotiations take place when the context user (e.g. an application) must specify minimal values for precision and accuracy which are acceptable for its purpose. Hence, the middleware infrastructure shall deliver only context information that satisfies such requirements.

4.5. Contextual Events

Contextual events represent abstractions of environmental situations and conditions that a context-aware system is interested in. A context event is specified in terms of context attribute values and predicates.

Contextual events are the basic elements for building asynchronous notifications and adaptations in context-aware systems. An application typically changes its behavior as a reaction to context changes specified by a contextual event. An example of a context change is the drop of a device's energy level. An application could be designed to react to such a change by subscribing to the corresponding contextual event, and thus avoiding the polling overhead that it would have been required to notice the context change.

In some cases, the condition that fires a contextual event is too complex to be described by the model, so it should be implemented by the context provider which is responsible for publishing the context information. For example, LIS uses a probabilistic algorithm to determine if a device has entered a symbolic area. For this reason the corresponding contextual event *EnteringInAnArea* cannot be easily specified as a conjunction of attribute values, so the firing of this event type is left to the LIS. Yet another example could be *proximity relations* such as *"user A is near user B"*.

Although contextual events are very useful abstractions, application-specific contextual events are still required for describing conditions which are restricted to an application domain. For example, conditions as *LowBatteryLevel* and *LowCommunicationBandwidth* might be interpreted differently, depending on the number of applications running in a device and application's resource requirements. Our programming model supports the adoption of application-specific contextual events as well.

4.6. Queries

Context queries represent the main interface between applications and the context infrastructure for synchronous type of access. Applications use context queries to retrieve the set of context information that adhere to some conditions. Our context model allows the specification of context-specific queries. For example, LIS supports several location-specific queries, such as *"retrieve all symbolic areas"*, *"retrieve the current symbolic area for a given device"* and *"retrieve all devices located in a symbolic area"*. In order to describe a query, a user must specify the query's abstract name, result type and its parameters.

Context queries allow the context modeler to devise specific semantics for queries, and which can be shared among all applications that need to use this context. In the same way as contextual events, some context queries are too complex to be defined using only general-purpose querying mechanisms. As for the events, also here the application developer may have not enough knowledge about context semantics, in order to describe queries correctly. Moreover, the use of context-specific queries may improve the overall performance of the system, since it might decrease the number of network interactions between an application and the context provider.

In our approach, retrieving context information using general-purpose query mechanisms remains possible. However, this should be restricted to queries for application-specific purpose.

5. Case Study: BuddySpaceLive

In order to demonstrate the concepts and modeling capabilities of our approach, this section describes a case study of an application called *BuddySpaceLive*, an extension of the *BuddySpace* [Knowledged Media Institute, The Open University 2005] open source application.

BuddySpace is an instant messaging (IM) client that enables location-based interactions by associating location information to each of the interacting users. Besides the usual interface of an IM application, *BuddySpace* allows the user to inform his current position within a map, that had been previously published in a map repository. The user's location is shared among all users that are in his buddy list (i.e. is shown in their maps). *BuddySpace* interacts with other IM applications through a Jabber server using the XML-based Jabber open protocol, which provides portability with most common IM platforms, such as MSN Messenger, Yahoo Messenger and ICQ.

In BuddySpaceLive (BSL), on the other hand, the user's location is obtained from MoCA infrastructure, which publishes this location as context information through its LIS service. Using SRM, BSL gets symbolic location of a user, instead of just the physical location, as provided by BuddySpace.

We have chosen as a testing scenario the 5^{th} floor of our Department's main building (RDC), shown in Figure 3. We adopted this test case because symbolic locations were well defined[1], we could reproduce consistent human movements, and the coverage of access points provided adequate location accuracy for LIS.

The following sections describe the BSL implementation according to three aspects: (1) the modeling of application and sensor-specific context information, described in sections 5.1 and 5.2, respectively; (2) the architectural design of services and the interaction among BSL instances and the Context Service, described in section 5.3, and (3) implementation of the context-dependent application logic, described in section 5.4.

[1]This floor contains professor's individual rooms and classrooms.

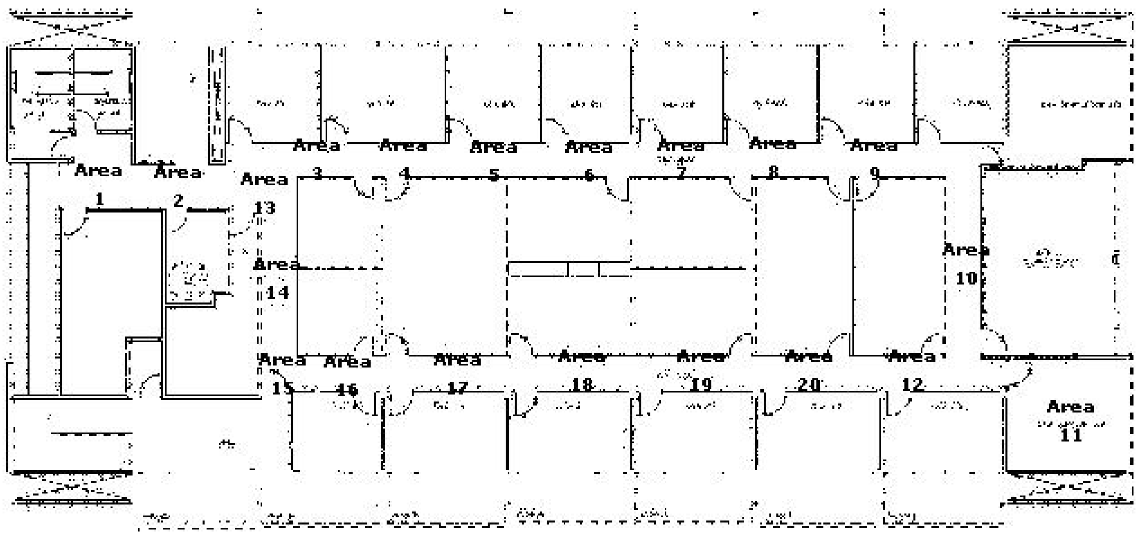

Figure 3. BuddySpaceLive map for the testing scenario

5.1. Application-level context modeling

In the first step of the application development, one should identify and model which context information the application should be aware of. Although most information that represents an application state can be modeled as a context, it is not always desirable to do so because of the incurred overhead of storing and handling context. Hence, an important task is to decide which application-specific information should be modeled and managed as context. In the case of BSL, the user profile (login, password, preferences) and user's buddy list are natural candidates for context. Unfortunately, current context modeling methodologies, such as the ones proposed in [Henricksen and Indulska 2006] do not cover this aspect.

In order to identify which of the application's state is eligible for being modeled as context, we have adopted a simplified criteria composed of the four aspects shown below, followed by the correspondent modeling decisions.

1. Identify the context-based adaptations required by the application for the currently provided context information → *location change*
2. Model transformations or interpretations of currently available context to application-specific domains → *interpret symbolic location as a location into a BSL map*
3. Identify which distributed state shall be shared among all application instances → *the location and the current device of each user*
4. Identify general-purpose state information → *mapping from device id to user id*

We have modeled BSL's context information in a context called `UserContext`, shown in Figure 4. This context information is comprised of user id, user status (e.g. online, offline), device id (MAC address of user's current device) and location. A BSL instance is responsible for publishing and updating the `UserContext` information. The instant messaging server could have been provider of such context, but we have preferred this modeling as it avoids having to change legacy software. Location modeling issues are discussed in next section.

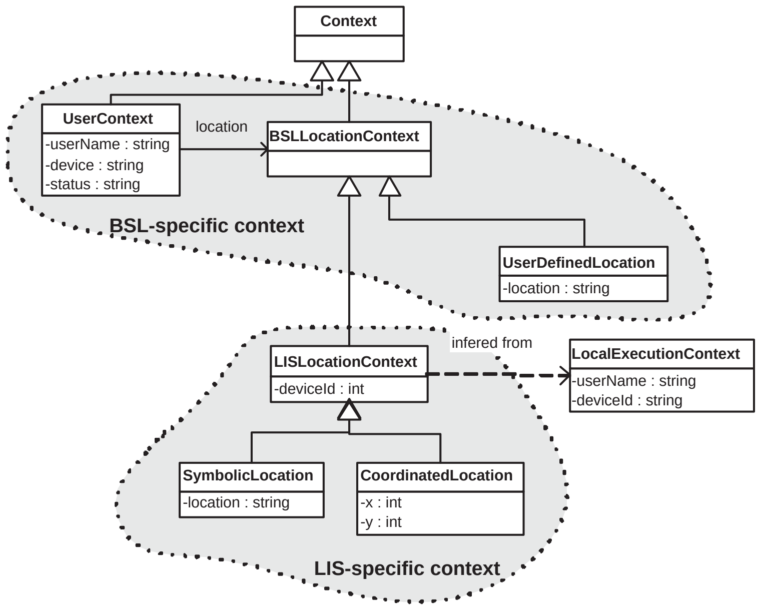

Figure 4. Context model for the BSL case

5.2. Sensor-level context modeling

At sensor-level context modeling, we have to investigate aspects of sensors limitations and QoC. In our case study, these aspects are related to the location information.

Since BuddySpace positioning is based on coordinates, we had to transform LIS symbolic locations into a coordinate-based location. For this, we used the central point of a symbolic region as the representative point for the user's location. This approximation is reasonable for our case because most regions have a central point that corresponds to a consistent location for user movement. For example, the floor's corridor is completely covered by consecutive symbolic regions which centers correspond to locations in the corridor (e.g. see *Area 1* to *Area 9* in Figure 3).

LIS publishes symbolic location with two QoC parameter: *precision*, which specifies the type/granularity of the symbolic information being informed (e.g. room, floor, building), and the *accuracy*, which is a numeric value representing the probability of correctness of the location information. A BSL user can change the precision requirement in order to specify the granularity of symbolic information he wants to receive. For a map of larger scale, users may want to change location precision to higher granularity. If the selected granularity is greater than the map (e.g. *building* precision for the testing scenario), then BSL shows the buddies at the corner side, or requests the user to select a map with less precision (e.g. a campus map).

The second type of location context is the *user-defined location* (`UserDefinedLocation`), as originally supported by BuddySpace, which holds location information that is given explicitly by the user. BSL must use this type of location if the user is in a region without 802.11 coverage.

Additionally, we modeled coordinate-based location as a derived context of `LisLocationContext` because, in fact, LIS infers physical location. However, since LIS does not export this information, `LisLocationContext` assumes the coordinates of the central point of a symbolic region, as before mentioned.

Figure 4 shows the these three context types. User-defined location is modeled as a type of BSL location, since this location context is not general-purpose, as opposed to the context provided by LIS. This modeling hides the actual location context types from the BSL developer, which only has to specify the suitable context precision for the map.

5.3. Architecture

Figure 5 shows the architecture of the solution, in terms of the interactions between context providers, context consumers and the context service. There are four main context providers: (a) MoCA's monitor, responsible for publishing device's context information; (b) LIS, responsible for publishing symbolic and physical location context; (c) SRM, responsible for publishing symbolic regions; and (c) the BSL instance, responsible for publishing BSL the user's status and the user-provided location, as shown in Figure 4.

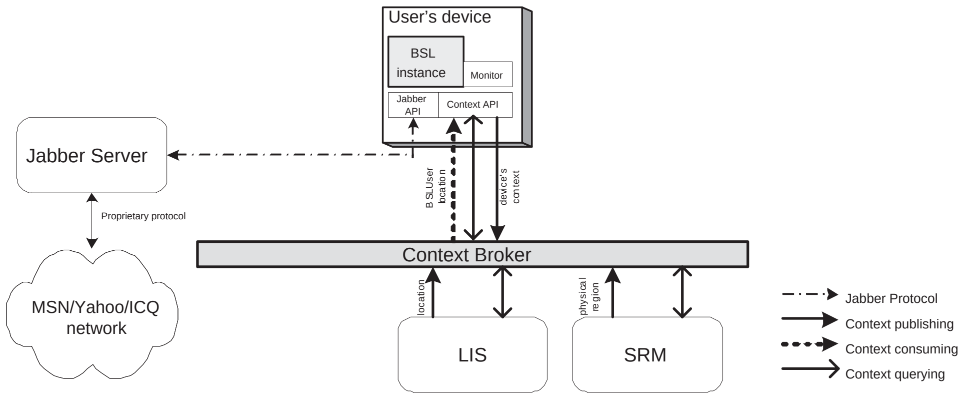

Figure 5. Architecture for BuddySpaceLive case study

In Figure 5, the *Context Broker* is an abstraction of the Context Service's components and its locations. The Context Service is transparent to the Jabber server, and therefore does not need to be modified in order to be integrated in the architecture. BSL instances receive contextual events as asynchronous notifications and interact with the context broker by the Generic Context Access API, a subset of MoCA middleware layer.

5.4. Application functionality

The functionality of BSL is implemented by handles for contextual events fired by the middleware and by synchronous context queries.

Since all BSL interactions use location context, they are part of the location context model. Although the Jabber protocol is in charge of updating buddies' status, the changes of their location can be implemented only by contextual events. Table 1 shows the contextual events for which a BSL user can register interest, and the corresponding application response at each event occurrence. In order to track user movements, BSL just needs to maintain triggers $E2$ for each buddy inside the map area and one trigger $E1$ for updating users that enter and exit the map region.

We have decided to register interest for buddy locations only if they are located in the current map. For the other buddies, the location is synchronously retrieved from

Contextual event	**Application response**
$E1$. User has entered/exited a region	Notify the user about a new buddy in the map Draw user location in the map and start updating his location
$E2$. User is moving	Update user location on the map
$E3$. User is near	Proximity indicator

Table 1. Some contextual events used in the BuddySpaceLive implementation

the Context Service using context queries. Table 2 describes some location-based queries used by BSL and the corresponding need in the application.

Query	**Application need**
$Q1$. Which users are in an area?	For establishing initial location of buddies on a map
$Q2$. Where is the user u?	Allows the user to identify the location of buddies outside a map
$Q3$. Which symbolic regions are available?	Register interest for events in a symbolic area Allows the user to switch location precision

Table 2. Context queries used in BuddySpaceLive implementation

The code below shows a simplified XML, covering the modeling of the $E1$ event and the $Q1$ query.

```
<context name="UserContext"
         base="Context" ... >
   <!-- context attributes -->
   <queries>
      <query name="usersInAnArea"
             resultType="UserContext">
         <param name="areaName"
                type="SymbolicLocation"/>
      </query>
      <!-- other queries -->
   </queries>
   <events>
      <event name="EnteringInAnArea" ... >
         <condition type="ProviderDefined" />
      </event>
      <!-- other contextual events -->
   </events>
</context>
```

BuddySpaceLive and other MoCA-based context-aware applications are available for download at MoCA's home page: `http://www.lac.inf.puc-rio.br/moca/`.

6. Related Work

Support for context-aware software development has been usually offered by frameworks, such as Context Toolkit [Dey et al. 2001], and middlewares, such as RCSM [Yau et al. 2002], Confab [Hong and Landay 2001] and PACE [Henricksen et al. 2005]. They fulfill differently the three requirements that motivated our work: support for heterogeneous environments, management of context evolution and support for context-specific abstractions. In this section we will focus our attention in comparing our approach with related work with respect to the latter aspect, since it have been more explored in this paper. For a complementary discussion, please refer to [da Rocha and Endler 2006].

Most of current research in context modeling is based on general-purpose modeling approaches which impose specific constraints on how applications access and use context information. For example, they do not consider abstractions for context changes, which are the basis for programming adaptations in context-aware applications. This restriction applies both to object-oriented based models, such as [Cheverst et al. 1999], and to ontology-based models, such as [Ranganathan and Campbell 2003]. In our opinion, it is difficult to use a general-purpose approach for supporting specific abstractions of a particular context-aware application domain.

Henricksen and Indulska [Henricksen and Indulska 2006] have proposed a framework for context-aware software engineering that offers a context modeling technique, an abstraction for describing context-based adaptations and a programming model for context-aware applications. This work is one of the most advanced proposals for integrating context modeling and software engineering. They propose a graphical approach for context modeling called CML (Context Modeling Language) that supports advanced modeling concepts, such as quality of information and ambiguity. This modeling approach also supports several kinds of associations among context types, such as derived association, for representing inference between two contexts. As discussed before, such association is not explicitly modeled in our approach.

In addition, Henricksen and Indulska proposed a very interesting abstraction, similar to our contextual events, called *situation*. A situation is expressed using predicate logic, combining expressions that define the necessary conditions for its occurrence. However, the authors propose a software infrastructure that maintains the modeled situations in an adaptation layer shared among all applications executing on a device. In contrast, we maintain context information and contextual events within the same management infrastructure, in order to reduce the number of required network communications between an application and the context management infrastructure. Another difference is that in their proposal the condition that fires a situation must always be explicitly modeled, whereas we allow complex conditions to be implemented through context providers.

Finally, Henricksen and Indulska adopt a generic layer for querying context. Consequently, they do not support abstractions for context-specific queries, unless they are implemented on the application side. In this case, however, this abstractions cannot the shared among applications in the context-aware system.

7. Conclusions

This paper has presented an approach for supporting context-aware applications based on a flexible context model and an infrastructure for evolutionary management of context information. The context modeling approach supports two context-specific abstractions: contextual events and context queries. We have argued that such abstractions are important for supporting a disciplined process of engineering context-aware systems, where common forms of context access and inference are programmed by the developer of the context providers rather then by the application programmer. However, such abstractions are still not sufficient. An architecture for context-aware programming should also provide complementary programming models, such as the branching model proposed in [Henricksen and Indulska 2006]. We are still investigating an extension of our architecture that adopts mobile agents for handling context information. Some of our future research efforts will be directed towards the development of a suitable programming model for context-aware development, as well as the extension of our modeling approach in order to support temporal queries and time-based conditions for event firing.

References

Capra, L., Emmerich, W., and Mascolo, C. (2003). CARISMA: context-aware reflective middleware system for mobile applications. *IEEE Transactions on Software Engineering*, 29(10):929–945.

Chan, A. T. S. and Chuang, S.-N. (2003). MobiPADS: a reflective middleware for context-aware mobile computing. *IEEE Transactions on Software Engineering*, 29(12):1072–1085.

Cheverst, K., Mitchell, K., and Davies, N. (1999). Design of an object model for a context sensitive tourist GUIDE. *Computers & Graphics*, 6(23):883–891.

Coutaz, J., Crowley, J. L., Dobson, S., and Garlan, D. (2005). Context is key. *Commun. ACM*, 48(3):49–53.

da Rocha, R. C. A. and Endler, M. (2005). Evolutionary and efficient context management in heterogeneous environments. In *MPAC'05: Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, pages 1–7, New York, NY, USA. ACM Press.

da Rocha, R. C. A. and Endler, M. (2006). Context management in heterogeneous, evolving ubiquitous environments. *IEEE Distributed Systems Online*, 7(4). art. no. 0604-o4001.

Dey, A. K., Abowd, G. D., and Salber, D. (2001). A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16(2, 3 & 4):97–166.

Henricksen, K. and Indulska, J. (2006). Developing context-aware pervasive computing applications: Models and approach. *Pervasive and Mobile Computing*, 2(1):37–64.

Henricksen, K., Indulska, J., McFadden, T., and Balasubramaniam, S. (2005). Middleware for distributed context-aware systems. *Lecture Notes in Computer Science*, 3760:846–863.

Hong, J. I. and Landay, J. A. (2001). An infrastructure approach to context-aware computing. *Human-Computer Interaction*, 16(2, 3 & 4):287–303.

Knowledged Media Institute, The Open University (2005). BuddySpace's home page. Available at: `www.buddyspace.org`. (Last visited: December, 2005).

Nascimento, F. N. d. C. (2005). A service for location inference of mobile devices based on IEEE 802.11. Master's thesis, Departamento de Informática, PUC-Rio. (in portuguese).

Raatikainen, K., Christensen, H. B., and Nakajima, T. (2002). Application requirements for middleware for mobile and pervasive systems. *SIGMOBILE Mob. Comput. Commun. Rev.*, 6(4):16–24.

Ranganathan, A., Al-Muhtadi, J., Chetan, S., Campbell, R., and Mickunas, D. (2004). MiddleWhere: a middleware for location awareness in ubiquitous computing applications. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 397–416, New York, NY, USA. Springer-Verlag New York, Inc.

Ranganathan, A. and Campbell, R. H. (2003). A middleware for context-aware agents in ubiquitous computing environments. *Lecture Notes in Computer Science*, 2672:143–161.

Sacramento, V., Endler, M., Rubinsztejn, H. K., Lima, L. S., Goncalves, K., and do Nascimento, F. N. (2004). MoCA: A middleware for developing collaborative applications for mobile users. *IEEE Distributed Systems Online*, 5(10).

Strang, T. and Linnhoff-Popien, C. (2004). A context modeling survey. In *First International Workshop on Advanced Context Modelling, Reasoning And Management*, Nottingham, England.

Yau, S. S., Karim, F., Wang, Y., Wang, B., and Gupta, S. K. S. (2002). Reconfigurable context-sensitive middleware for pervasive computing. *IEEE Pervasive Computing*, 1(3):33–40.