Synthesizing Context for a Sports Domain on a Mobile Device

Alisa Devlic[1,2], Michal Koziuk[3], and Wybe Horsman[4]

[1] Appear Networks, Kista Science Tower,
16451 Kista, Sweden
alisa.devlic@appearnetworks.com
[2] Royal Institute of Technology (KTH), Department of Communication Systems,
Electrum 418, SE-164 40 Kista, Sweden
devlic@kth.se
[3] Institute of Telecommunications, Warsaw University of Technology,
ul. Nowowiejska 15/19, 00-665 Warsaw, Poland
mkoziuk@tele.pw.edu.pl
[4] Capgemini NL bv, Papendorpseweg 100
3528BJ Utrecht, The Netherlands
wybe.horsman@capgemini.com

Abstract. In ubiquitous computing environments there are an increasing number and variety of devices that can generate context data. The challenge is to timely acquire, process, and deliver these data to context-aware applications. The role of context synthesis is to generate new knowledge, as a result of a reasoning process applied to context information that is already present in the system. The success of this mechanism mainly depends on the response time that the end-user or an application must wait for the response to a context query. This paper describes and evaluates an approach to context synthesis on a mobile device to be used by a set of applications in a sports domain. A scenario based on a live race at the Super Prestige Cyclocross in Gieten, Netherlands demonstrates the use of context synthesis to dynamically compose gaps and groups of cyclists in order to provide a nearly real-time virtual ranking service.

Keywords: context synthesis, context operators, context modeling, sport scenario.

1 Introduction

Imagine experiencing a sport event from your phone, where you are your own director deciding upon your own point of view by actually moving about the event locale. Rather than simply selecting one of many video streams on your screen, instead you utilize the abstract view of the event (as viewed on your smartphone) to select your own personal viewpoint of the event. Therefore, you want answers to questions, such as: what are the positions of the Rabobank riders, what gaps and groups of riders are there on the track today, who is the virtual leader of the race at this moment, at what time can I expect the leader to pass my current position on the track, etc. Based upon

© Springer-Verlag Berlin Heidelberg 2008

the answers to these questions you will move to the position which you decide will give you the best vantage point.

It is November 2007 in Gieten, Netherlands. It is cold and rainy, the perfect conditions for an international cyclo-cross race. The track lies partly in the woods and partly around a pond with steep banks. The track is about 3 km in length and each rider must complete 7 laps. The difference between the top riders and the ones that have a bad day is very big; after 2 laps the slowest rider is lapped by the fastest, but by then most distinguishing features of the riders are covered with mud – thus after 3 or 4 laps it is hard to see who is who. Using devices with the MIDAS middleware and race application preinstalled, attendees are able to see their own location and the location of the individual riders on a map of the track, the leading cyclist in the race, the total and remaining distance in meters, the gap between riders, as well as which riders are riding in the same group (so called gaps and groups analysis). All of this is shown live on your mobile device.

MIDAS (*Middleware Platform for Developing and Deploying Advanced Mobile Services*) [1] is a European research project concerning 3G and beyond, which aims to define and implement a platform to simplify and speed up the task of developing and deploying mobile applications and services. It is specifically designed to be used in MANETs. MIDAS enables applications running on different nodes to share information by inserting data in and retrieving data from a shared data space. This shared data space is implemented using a combination of data replication and remote operations – but this fact is transparent to applications. Therefore, for the purpose of this paper, we assume that all context information is available locally on a mobile device.

An application using this middleware calculates and displays gaps and groups of cyclists in near real time. This calculation needs to be performed as the cyclists' relative positions change, resulting in the synthesis of gaps and groups context. Moreover, the presentation on the display needs to be updated to reflect the current composition of groups. Thus, the middleware periodically obtains cyclists' geographic locations and utilizes information about the waypoints in the race. This data can be combined with the cyclists' data (such as name, team name, identification number, etc.), in order to perform context synthesis.

This newly generated context information can be in turn used by multiple applications. Hence, applications requesting customized context may share the cost of producing this synthesized context information. Additionally, each application need not be concerned with how this synthesis is implemented. However, some applications may want to implement their own synthesis functions. We refer to these functions as *context operators*. Context operators enable different applications and even different context-aware systems in the same domain to query each other about the context information which could be synthesized using the functions they implement. For example: a racing application and media application deployed on different devices should be able to remotely query each other (using the same middleware API and context operators) for results of the race and rankings of all athletes in the competition. The output of the operator is sent as a result of a context query. This result is called a *synthesized context*, since it was generated by context synthesis.

Our motivation and the idea for context synthesis using operators was previously presented in [2]. The main advantages of our approach are increased reusability, extensibility, and interoperability, facilitated by context operators and exploiting ontology based context modeling. This paper describes the realization of this approach via a race application developed on this middleware, and evaluates the context synthesis in terms of the response time to a context query sent by the application. The response time is divided into the time needed to find the correct operator, the time needed to obtain context information (formatted as ontology data) from its repository, and the time needed by this operator to perform the actual context synthesis.

The rest of the paper is organized in seven sections. Section 2 elaborates the MIDAS context modeling approach using ontologies for mobile devices. Section 3 presents our approach for context synthesis using context operators, while Section 4 describes the set of applications developed for sports scenario that illustrate the use of context operators for context synthesis. In Section 5 we give a performance evaluation. Section 6 provides a brief overview of related work. We conclude in Section 7 with a summary of the results and plans for future work.

2 Context Modeling Using Ontologies

In order for MIDAS to be a context aware framework it needs a mechanism for modeling and representing context. This context model must contain information specific to a specific domain of deployment of a MIDAS based service. A context model of a domain describes the people, objects, and relations between them which are typically encountered in a specific situation (or types of situations). Focusing only on a single domain makes it possible to create a very specific model, capable of representing a very fine level of detail, which otherwise could not be captured due to growing complexity of a more general model.

The context model used is an ontology, which is provided to the system in the form of a file. Thus the same middleware can be used in various domains given a new ontology file. We envision that an organizer of an event creates an ontology which represents the domain of this event. This ontology is provided to application developers (who can create applications for this particular domain). Once this ontology is created, other similar events can re-use the ontology, modifying it as required.

The ontology language chosen for the domain model is DL-Lite [3], which is a subset of OWL-DL optimized for fast reasoning on top of relational databases. This language supports the basic terms of classes and properties, and it handles statements about subsumption, disjointness, role-typing, participation constraints, non-participation constraints, and functionality restrictions. MIDAS implements an architecture for handling DL-Lite ontologies on a Java enabled mobile device [4].

The decision to use DL-Lite as a language for MIDAS ontologies was motivated by the results obtained during an initial experiment [5]. This experiment showed that using OWL-DL [6] with existing off the shelf solutions such as Jena [7] and Pellet [8] could not be applied on mobile devices, given their poor (slow) performance even on desktop machines and very high memory requirements. The use of existing ontology query languages, such as RQL [9] or SPARQL [10] was not analyzed. However, these solutions are usually not designed for mobile devices as their main focus is high expressivity. Thus, their practical usability in a mobile device setting is unlikely. We chose DL-Lite because of its relative simplicity and optimization for fast performance. The limitations in the description logic that made these improvements possible turn out not to be limiting when modeling a domain.

The syntax chosen for context model ontologies is the Manchester OWL Syntax [11] due to features which make it more suitable for applications on mobile devices than the usual OWL Syntax [12]. The main feature is that it is much easier to parse, as it requires only two linear scans of the ontology file, and does not require construction of a tree structure during parsing. Another feature is that an ontology is approximately half the size of the equivalent OWL representation, and because it is human readable it is possible to edit it by hand if necessary.

For representing the ontology on a MIDAS enabled mobile device we created a dedicated Lightweight Ontology library [13], which implements the Jena [7] API in a form suitable for mobile devices. This library parses the ontology file and creates an in-memory representation of the ontology (supporting all the structures present in OWL-DL) based on HashTables. Its simplicity suits resource constrained devices (such as J2ME mobile phones).

The scenario examined in this paper is a cycling race. Part of the context model ontology is shown in Fig. 1. This example shows only the part of the class hierarchy from the domain model, which contains classes corresponding to roles of users. Other classes (not shown) in the domain model are used to represent places encountered during the event, abstract entities such as a group of cyclists, a gap between two groups, etc.

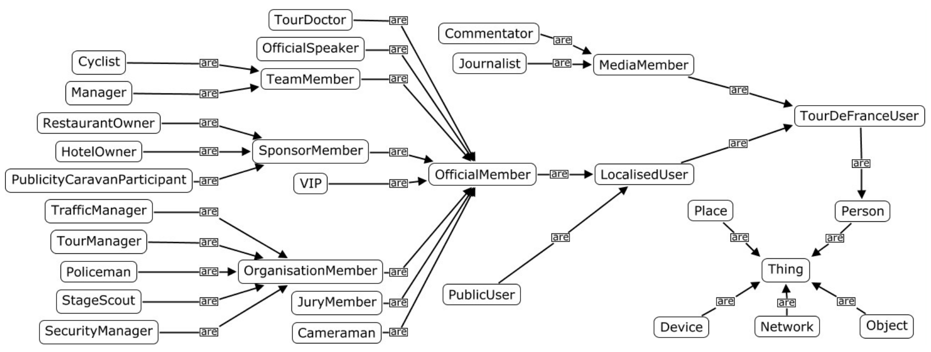


Fig. 1. Classes that describe roles of users involved in the cycling event

We consider five types of entities, which can be characterized as owners of context information: a person, a device, a network, a place, and an object. However, these entities are not independent, but have the following relations (see Fig. 2): a device is connected to a network; a person uses certain device(s); a person, device, and an object may be located at a place; a person and a device are somehow related to some other object(s). All entities are subclasses of the root class “Thing” in the ontology, from which all other terms are derived. Thus, we assign all context information to a certain entity and we can query information about an entity—i.e., user context, device context, network context, place context, and object context.

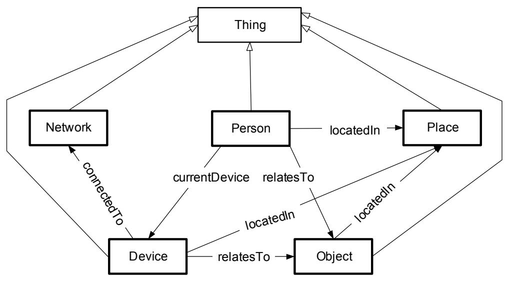


Fig. 2. Context entities and their relations in the context model

The context modeling architecture is implemented by a Context Knowledge Base component in the middleware. The API offered by this component makes it possible to model context by means of objects of the type *DomainInstance*, each of which represent physical entities. A *DomainInstance* can be added or removed as needed. Each *DomainInstance* object can have a number of property values assigned to it, and can belong to a number of classes. These classes are represented by objects of the type *DomainClass* and *DomainProperty* (respectively) which correspond to those present in the domain model ontology.

Context information needs to be stored by the middleware before it can be queried or synthesized. In order to store, retrieve, and manipulate the formatted (higher-level) context information, we developed a means of mechanically mapping the domain classes from the context model to the corresponding java classes, as well as from property names to java class variables.

3 Context Synthesis Using Context Operators

Operators for context synthesis are domain-specific functions over the context data. The benefits of these operators are that by performing operations over the existing context information, new context information that previously did not exist in the system can be produced. The output of the operation performed by an operator, a *synthesized context*, is sent to the application as a result of a context query. Operators could be used on a higher level to synthesize information of a certain user, device, network, place, or other object, as illustrated earlier in Fig. 2.

Operators are bundles of both a description and implementation; and described by an ontology, similar to the representation of context. They are implemented as java scripts that perform an action specified in the operator's ontology. The operator's description specifies the name of this operator, the types of the required input arguments, the returned output type, and the list of other operators used in performing the operator's function. As with the context model, operators are created for a specific domain and can be used by a set of applications in that domain. In order to provide context synthesis functions for applications in another domain, a new set of operators needs to be provided to the middleware, along with their ontology schema.

We distinguish between generic and specialized operators. Generic operators are part of an ontology schema, representing an umbrella for all the different implementations of a function they provide. They are also part of an API provided to application developers. On the other hand, specialized operators can be created/modified and inserted into the middleware by application or system developers. Specialized operators are not directly visible to application developers; which operator is invoked will be determined by the middleware at runtime.

Specialized operators are implemented as scripts using Beanshell [23], an open source java script engine. In our implementation the operator scripts are part of the context service process and they can be programmatically added and removed by the middleware.

Fig. 5 shows the structure of the *Operator space* – a repository of operators. The root folder (i.e. operators/) contains all generic operators (which are also folders), containing in turn their specialized operators. Note that specialized operators are bundles of an operator description (an instance of the operator ontology encoded in Manchester OWL format, i.e., a .man file) and an operator implementation (a java script written in Beanshell, i.e., a .bsh file).

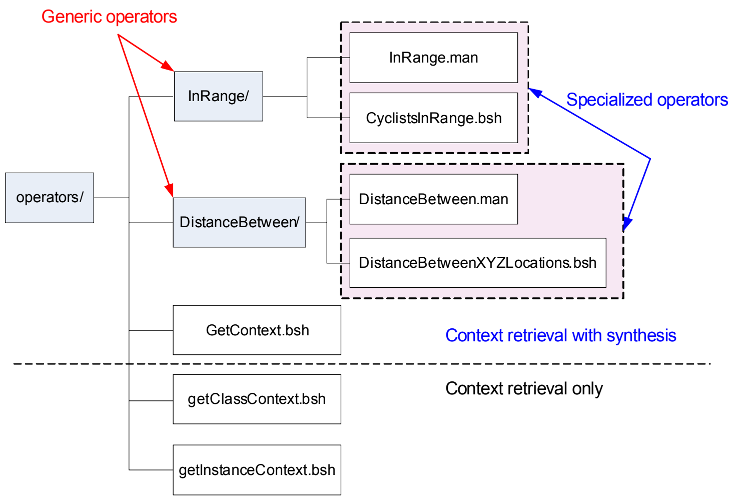
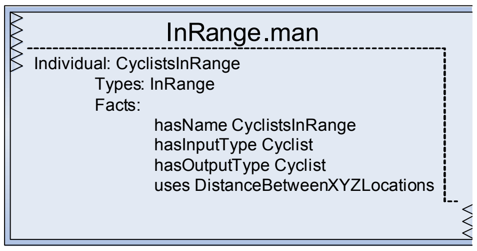


Fig. 3. Operator space file structure

The root folder of the *Operator space* shown in Fig. 3 also contains three specific operator scripts which are responsible for retrieving context data of the specified context owner, from the Context knowledge base: *GetContext.bsh*, *GetClassContext.bsh*, and *GetInstanceContext.bsh*. Note that they do not have a generic operator representing them, and they are used for distinct purposes. As previously noted, when specific context operators need to retrieve context, they will provide *DomainInstance* objects to the *GetContext* operator to retrieve the missing context values. It is also possible to retrieve context data directly from the repository without context synthesis, via the *GetClassContext* and *GetInstanceContext* scripts. *GetInstanceContext* is used to obtain the domain instance with the supplied datatype properties from the context query.

We can also query the Context knowledge base for other properties of the same instance. *GetClassContext* is used when we do not know the instance, but rather use a domain class with the specified property name-value pair to identify this instance.

An example of an operator description file, *InRange.man* is presented in Fig. 4. This file contains all the specialized operator descriptions. Fig. 3 shows only *CyclistsInRange*, but there could be others as well (e.g., *UsersInRange*). The description of the *CyclistsInRange* (specialized) operator is interpreted in the following way: it has the name "*CylistsInRange*" and is derived from a generic operator (i.e. *InRange*). It requires an input of the type Cyclist and produces an output value of the type Cyclist. The operator uses the result from another (simpler) operator *DistanceBetweenXYZLocations* to calculate the distance between two locations.



```
InRange.man
Individual: CyclistsInRange
          Types: InRange
          Facts:
                    hasName CyclistsInRange
                    hasInputType Cyclist
                    hasOutputType Cyclist
                    uses DistanceBetweenXYZLocations
```

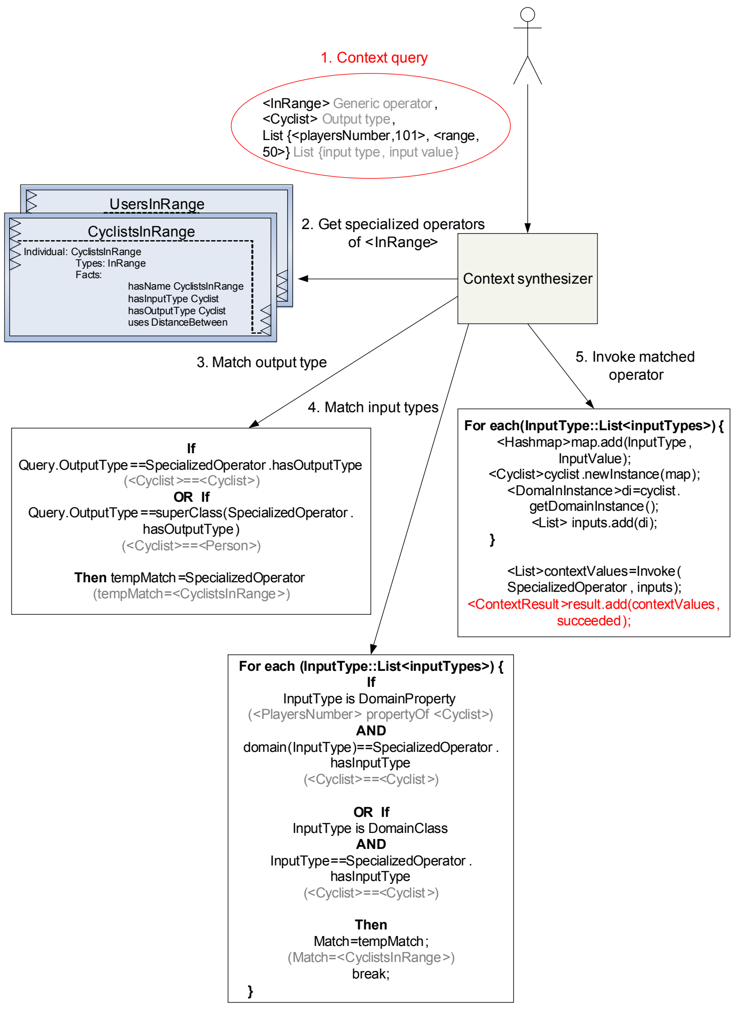
Fig. 4. CyclistsInRange description

In order to generate this description file, the developer needs to programmatically set the type of this specialized operator, the list of input types, the output type, as well as operator dependencies. The middleware will automatically append this operator description to the correct ontology file (if this file does not exist, it will be created in the correct location).

Note that all specialized operator scripts take as inputs *DomainInstance* objects, which are instances of classes specified as input types in their operator description file. Thus these domain instances pass the input arguments from the context query to the operator's method, and can be used to retrieve the missing information from the Context knowledge base (if needed).

3.1 Operator Matching

The context synthesizing process determines the most appropriate specialized operator to invoke from the available (specialized) operators by using a reasoning process (which takes into account the required output type and supplied input types). The idea behind the operator matching algorithm, illustrated in Fig. 5, is to enable different applications (or even different context systems) in the same domain (in our scenario a sport domain) to use the same "functions" to synthesize context information, without being concerned about the implementation of these functions. The operator matching algorithm returns the specialized operator with either exactly the same description as specified by the query or a more generic one.



```
If
Query.OutputType==SpecializedOperator.hasOutputType
(<Cyclist>==<Cyclist>)
OR If
Query.OutputType==superClass(SpecializedOperator.
hasOutputType)
(<Cyclist>==<Person>)

Then tempMatch=SpecializedOperator
(tempMatch=<CyclistsInRange>)
```

```
For each(InputType::List<inputTypes>) {
    <Hashmap>map.add(InputType,
        InputValue);
    <Cyclist>cyclist.newInstance(map);
    <DomainInstance>di=cyclist.
        getDomainInstance();
    <List> inputs.add(di);
}

<List>contextValues=Invoke(
    SpecializedOperator, inputs);
<ContextResult>result.add(contextValues,
    succeeded);
```

```
For each (InputType::List<inputTypes>) {
    If
    InputType is DomainProperty
    (<PlayersNumber> propertyOf <Cyclist>)
    AND
    domain(InputType)==SpecializedOperator.
    hasInputType
    (<Cyclist>==<Cyclist>)

    OR If
    InputType is DomainClass
    AND
    InputType==SpecializedOperator.
    hasInputType
    (<Cyclist>==<Cyclist>)

    Then
    Match=tempMatch;
    (Match=<CyclistsInRange>)
    break;
}
```

Fig. 5. This figure shows the algorithm itself, initiated by the user's context query, along with the invocation of the matched (specialized) operator

An example of a context query is: *InRange(”101”, 50, ModelConstants.Cyclist),* where the response time is bounded to *5sec*. This example can be interpreted as follows: retrieve all cyclists in the range of 50 meters from the cyclist with the ID=”101” and the result should be returned within 5 seconds. If the result is not computed by that time, the synthesis process will be interrupted, and a response will be returned to the query initiator containing an empty list of values and a flag indicating that the query was unsuccessful. After receiving the query, the operator matching algorithm retrieves all available specialized operators and processes the supplied data in order to find an exactly matching specialized operator (by checking if output and input types of the operator and the query match). Otherwise it will return a more generalized one, i.e. *UsersInRange*, which would return *Users* instead of *Cyclists* as result. Finally, it invokes the matching operator.

4 Cyclist Race Application

The cyclist race application set consists of a number of applications responsible for: 1) entering static cyclist data and managing track waypoints, 2) processing and showing a list of the latest rider location data and 3) showing the actual gaps and groups of cyclists to the end user during the race. The last (end-user) application is available with a user interface in three different form factors for display on a small device (Nokia N800), a laptop (HP tablet), and as a side bar next to a web page shown on a laptop. The processing application was at the time of the race not actually deployed on a mobile device; however, in Section 5 we give an evaluation of context synthesis on a Nokia N800.

The geographic locations of the cyclists are obtained from GPS receivers attached to cyclists' arms and this context is synthesized into *gaps and groups* information. The *gaps and groups* information is in turn broadcasted to all interested users equipped with the MIDAS middleware and an end-user application installed on their devices. The frequency of updates is about once every three seconds. A video demonstrating the live race at the Super Prestige Cyclocross using MIDAS middleware and the described application set can be seen at [21].

The gaps between groups of cyclists and the composition of groups are synthesized from the following cyclists' context: the last known cyclists position information, the roadbook waypoints, and the configured maximum distance between two consecutive cyclists of one group. A gap is defined as a distance between locations of two consecutive cyclists that exceeds a predefined threshold. In cycling a distance of about 25 meters is considered a gap. Cyclists between two consecutive gaps compose a group. In order to calculate gaps and groups, the application needs to calculate the distance between the successive waypoints and their distance to the finish (based on Haversine's Formula [14] combined with John P. Snyder's curvature [15]).

Figure 6 illustrates the operators used to calculate gaps and groups information. In order to improve the performance, all real time objects are stored in the memory. To share these objects between applications singleton instances are stored in the operator space running environment; therefore an operator has to be used to interact with these objects. Moreover, the output of one operator is used as an input to the other one.

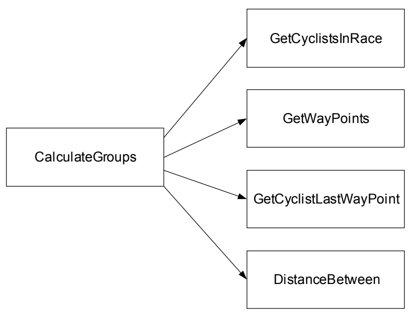
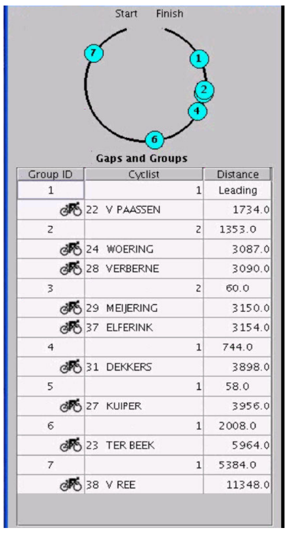


Fig. 6. Set of operators used in the application

To calculate the gaps and groups, the algorithm exploits the fact that every cyclist cycles from one waypoint to the other and sends several GPS measurements while on this path to reach the waypoint. Based on received GPS measurements, the algorithm computes location, distances between cyclists, their order, and if the distance between two cyclists is 25 meters or greater, then there is a gap.

Groups in the race are presented graphically to the user via a user application, as shown in Fig.7. The circle represents the whole track. Each dot represents a group. The progress of the groups is shown relative to Start and Finish. Additionally, the list of groups is presented to the user as a textual table. The first column of the table contains the group name (1 to n), the second column shows the number of cyclists present in the group, and the third one contains the distance to the preceding group. The leading group distance is replaced with a "Leading" indicator. For every cyclist, the first and last name, player number, as well as distance to the finish are shown. Once a group finishes the race, the distance is replaced by a "Finished" indicator, the cyclist icon is replaced by a flag, and the line is printed in green.



Gaps and Groups

Group ID	Cyclist	Distance
1	1	Leading
	22 V PAASSEN	1734.0
2	2	1353.0
	24 WOERING	3087.0
	28 VERBERNE	3090.0
3	2	60.0
	29 MEIJERING	3150.0
	37 ELFERINK	3154.0
4	1	744.0
	31 DEKKERS	3898.0
5	1	58.0
	27 KUIPER	3956.0
6	1	2008.0
	23 TER BEEK	5964.0
7	1	5384.0
	38 V REE	11348.0

Fig. 7. Application GUI with actual data

5 Performance Evaluation

The MIDAS middleware and applications are implemented in Java. We ran all performance tests on a Nokia N800 device with the JamVM virtual machine [22] with a compiler for Java 1.4. This device was chosen by the MIDAS project because it is Linux based, allowing network and low-level programming. We also used a third party library for implementation of java scripts, Beanshell [23].

The performance of context synthesis is evaluated in terms of the response time of operator matching, context retrieval, and context processing (i.e., operator invocation),

Table 1. Response times

Average response times with varying number of specialized operators (i.e., 1, 2, 5, 10)	Based on 10 **first** queries	Standard deviation (based on 10 **first** queries)	Based on 10 **subsequent** queries	Standard deviation (based on 10 **subsequent** queries)
Matching algorithm time	2.49 sec	0.009 sec	1.94 sec	0.07 sec
Loading specialized & root scripts time	1.7 sec	0.087 sec	No average, for the first time only (1.7 sec)	No standard deviation
Total operator matching time	4.2 sec	0.087 sec	1.94 sec	0.07 sec
Context retrieval time	0.37 sec	0.006 sec	0.09 sec	0.001 sec
Loading dependency scripts time	0.15 sec	0.001 sec	0.17 sec	0.015 sec
Operator invocation time	0.67 sec	0.008 sec	0.36 sec	0.04 sec
Total query time	5.4 sec	0.045 sec	2.57 sec	0.07 sec

as well as the overall response time to a query sent by an application. The values shown in Table 1 were obtained by sending the same context query, but varying the number of available specific operators (i.e., 1, 2, 5, and 10) when performing the operator matching algorithm, and then calculating the mean value.

Note that before the java scripts can be invoked, they have to be loaded into the interpreter and the classpath has to point to the folder where these scripts reside. These scripts can also invoke other scripts (from different folders), thus these other scripts need to be invoked in the caller's context (the so called *namespace*). Therefore, when the first query is sent, the total time needed to find the most appropriate specialized operator (i.e., the *total operator matching time*) also includes the time needed to set the namespace to point to the generic operator folder (e.g., *InRange*), as well as load specific operator scripts from this folder and from the root operator folder. For all successive queries this operation is cached. When invoking the specialized operator found by the matching algorithm, some additional time is needed to load the scripts from the dependency operator folder (e.g., *DistanceBetween*).

As it can be seen from Table 1, the response times for the first query are twice as large as for the other following queries, because the caching speeds up the subsequent operations. The operator matching algorithm takes 2 seconds on average, however for the first query it requires 4 seconds (including the initial time needed for loading the necessary scripts). Context retrieval (of three cyclists' data) was rather quick as was the operator invocation time. The number of concepts required by an application was small. With regards to performance with increasing number of domain instances, please refer to [4]. Note that operator invocation time includes the time needed to invoke *CyclistsInRange* and *DistanceBetween* operators. We used SQL prepared statements to retrieve context from an HSQL database. The total time needed to receive the result of context query took on average 2.5 seconds, but 5.4 seconds for the first query.

Note also that in some other scenario it could happen that after the second query the first query is made again but containing some other operator, this will also require operator matching. However, we plan (as future work) to introduce caching of queries and matched specialized operators in order to reduce the total query time.

There were 1000 spectators along the race course. Note that this deployment was intended as a proof of concept to validate middleware functionalities and was not designed to be an evaluation of the system using a statistically significant number of users. However, the impression of 9 users (monitoring the race on 6 tablet PCs and 3 Nokia N800 devices) was very positive. A few seconds of delay did not affect their "near real-time experience". Furthermore, users liked the way that they could select their favorite cyclists in the application, in order to know when he/she will pass their location. Zooming functionality also helped to improve overcome the limitations of the small screen when more cyclists were tracked during the race.

So far we have not examined the cases when context changes rapidly nor we have considered the issues concerning uncertainty in the context. We plan to address these issues in future work.

6 Related Work

Our context synthesizing work was inspired by the Aura Contextual Information Service (CIS) research project [16]. However, our queries are not SQL-like, but instead they are object-oriented, containing context operators which perform synthesizing operations. Context operators can in turn use other simpler operators to execute smaller tasks and to reuse existing functionality.

Modeling context with ontologies is in itself not a novel idea. Surveys of context modeling frameworks clearly indicate that modeling context with ontologies is the most expressive way to do it [17]. Typically, mobile devices being part of a context aware system need to remotely access the ontology model, and the context data. In case of CoBrA [18] the remote facility is an ‘intelligent agent’, called a Context Broker, which acts as a central point of the system maintaining a representation of context common to all the devices in the network. The SOCAM [19] solution also relies on a shared context space located on an external device (an OSGi gateway) which can be accessed by multiple context aware services. MIDAS differs from these architectures in that it is capable of handling ontologies on mobile devices, which makes it possible to provide local access to context modeled with ontologies for every device in the network. This seems especially useful in ad-hoc networks where access to a central server cannot be provided.

J. I. Hong and J. A. Landay [20] emphasize a need for creating a basic infrastructure services and application-specific services, the latter implemented on a case-by-case basis. One such basic infrastructure service is automatic path creation, which transforms raw sensor data to higher-level context data. It automatically composes operators based on high-level needs and what resources are available. Our work extends this idea to enable multiple applications or even different context-aware systems to use the same operators designed for a specific domain without being concerned about their implementation. Moreover, we also enable chaining of operators, where each operator takes some existing context information (as defined by a context model) as input and provides new context information as an output. All applications can reuse already deployed operators and add their own implementations of the same generic operators.

7 Conclusion and Future Work

We have presented and evaluated the approach for context synthesis using operators on a Nokia N800 device. Operators for context synthesizing are domain-specific functions over the context data. The benefits of these operators are that by performing operations over the existing context information, new context information that previously did not exist in the system can be produced. Moreover, applications can use the same operators to synthesize context information, without being concerned about their implementation. This also enables applications to share the cost of context synthesis by querying about the result of operators invocation.

We have evaluated this operator-based context synthesis approach in terms of response time to context query sent by the application and showed that it is possible to perform context synthesis operation in near real time (i.e., with the average latency of 2 seconds) on the mobile device. The main advantages of context operators are the reusability, extensibility, and interoperability, facilitated by ontology-based context modeling. For this purpose MIDAS provides a dedicated Lightweight Ontology library for representing and manipulating ontologies on mobile devices. We also demonstrated the use of context operators in the cyclist race application.

We plan to evaluate the response time of executing the remote operator invocation as well as to use caching decisions made by operator matching algorithm for a certain context query. We will also conduct a user study based on our next deployment. As a next step in context synthesizing we plan to use operators to combine inference algorithms in order to derive about high-level context.

Acknowledgements. The authors of this paper would like to acknowledge the partial financial support given to this research by the EU IST MIDAS project (6th Framework Programme, contract number 027055). We would also like to thank Prof. Gerald Q. Maguire Jr. for his valuable comments to this research work.

References

1. EU FP6 IST MIDAS project (2008), `http://www.ist-midas.org`
2. Devlic, A., Klintskog, E.: Context retrieval and distribution in a mobile distributed environment. In: Third Workshop on Context Awareness for Proactive Systems (CAPS 2007), Guildford, UK (2007)
3. Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Rosati, R.: DL-Lite: Tractable description logics for ontologies. In: 20th National Conference on Artificial Intelligence (AAAI 2005), Pittsburgh, Pennsylvania, USA, pp. 602–607 (2005)
4. Koziuk, M., Domaszewicz, J., Schoeneich, R.O.: Mobile Context-Addressable Messaging with DL-Lite Domain Model. In: The 3rd European Conference on Smart Sensing and Context (EuroSSC 2008), Zurich, Switzerland, October 29-31 (to appear, 2008)

5. Domaszewicz, J., Koziuk, M., Schoeneich, R.O.: Context-Addressable Messaging with ontology-driven addresses. In: The 7th International Conference on Ontologies, DataBases, and Applications of Semantics (ODBASE 2008), Monterrey, Mexico (to appear, 2008)
6. Smith, M.K., Welty, C., McGuinness, D.L.: OWL Web Ontology Language Guide. W3C Recommendation (2004), `http://www.w3.org/TR/owl-guide/`
7. Carroll, J.J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., Wilkinson, K.: Jena: Implementing the semantic web recommendations. Technical Report HPL-2003 (2003), http://citeseer.ist.psu.edu/carroll04jena.html
8. Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. Web Semantics: Science, Services and Agents on the World Wide Web 5(2), 51–53 (2007)
9. Magkanaraki, A., Alexaki, S., Christophides, V., Plexousakis, D., Scholl, M., Tolle, K.: RQL: A Functional Query Language for RDF. In: Gray, P.M.D., Kerschberg, L., King, P.J.H., Poulovassilis, A. (eds.) The Functional Approach to Data Management: Modelling, Analyzing and Integrating Heterogeneous Data. LNCS. Springer, Heidelberg (2004)
10. Sirin, E., Parsia, B.: SPARQL-DL: SPARQL Query for OWL-DL. In: Proceedings of the OWLED 2007 Workshop on OWL: Experiences and Directions, Innsbruck, Austria, June 6-7 (2007)
11. Horridge, M., Drummond, N., Goodwin, J., Rector, A., Stevens, R., Wang, H.: The Manchester OWL Syntax. In: OWL: Experiences and Directions 2006, Athens, Georgia, USA (2006)
12. Patel-Schneider, P.F., Hayes, P., Horrocks, I.: OWL Web Ontology Language Semantics and Abstract Syntax. W3C Recommendation (2004), `http://www.w3.org/TR/owl-semantics/`
13. Jabłonowski, M., Boetzel, P.: Middleware Layer For Semantic Object Tagging. Master Thesis at Warsaw University of Technology, Warsaw, Poland (2007)
14. Sinnott, R.W.: Sky and Telescope. Virtues of the Haversine 68(2), 159 (1984)
15. Snyder, J.P.: Map Projections – A Working Manual., U.S. Geological Survey, Professional Paper 1395, US Government Printing Office, Washington DC (1987)
16. Judd, G., Steenkiste, P.: Providing Contextual Information to Pervasive Computing Applications. In: First IEEE International Conference on Pervasive Computing and Communications (PerCom 2003), Fort Worth, Texas, pp. 133–142 (2003)
17. Strang, T., Popien, C.L.: A context modeling survey. In: Workshop on Advanced Context Modeling, Reasoning and Management as part of the 6th International Conference on Ubiquitous Computing (UbiComp 2004), Nottingham, England, pp. 33–40 (2004)
18. Chen, H., et al.: A Context Broker for Building Smart Meeting Rooms. In: Proceedings of the Knowledge Representation and Ontology for Autonomous Systems Symposium, 2004 AAAI Spring Symposium, Palo Alto, CA, USA (2004)
19. Gu, T., Wang, X.H., Pung, H.K., Zhang, D.Q.: An Ontology-based Context Model in Intelligent Environments. In: Proceedings of Communication Networks and Distributed Systems Modeling and Simulation Conference, San Diego, California, USA (2004)
20. Hong, J.I., Landay, J.A.: An Infrastructure Approach to Context-Aware Computing. Human-Computer Interaction 16(2, 3, 4), 287–303 (2001)
21. MIDAS video (2007), `http://www.youtube.com/watch?v=yulUmlVH8Jc`
22. JamVM – A Compact Java Virtual Machine (2008), `http://jamvm.sourceforge.net/`
23. Beanshell - Lightweight scripting for Java (2008), `http://www.beanshell.org/`