Implementation of Telematics Services with Context-Aware Agent Framework

Kenta Cho[1], Yuzo Okamoto[1], Tomohiro Yamasaki[1], Masayuki Okamoto[1], Masanori Hattori[1], and Akihiko Ohsuga[1]

TOSHIBA Corporation
1 Komukai-Toshiba-cho, Saiwai-ku, Kawasaki-shi, 212-8582, Japan
kenta.cho@toshiba.co.jp

Abstract. With the development of a car-mounted navigation system and an in-car network, improved information services for creating drive plans automatically have been realized. We propose a method of developing telematics services through combination of a context-aware agent framework and a planner that provides the predicted feature situation to the framework and creates drive plans composed of POIs (Points Of Interest) proposed by the framework.

1 Introduction

Context-aware applications are attracting attention as an approach to distinguishing important data from a large amount of contents spread on a network. A context-aware application recognizes the user's situation and offers information that may be useful for the user in that situation. Telematics, that is, advanced information services based on in-car embedded devices and networks, has become widely used. Services providing information in a mobile environment are realized with a network enabled car navigation system [1] [2]. When using a car navigation system to create drive plans from a starting point to a destination, user have to select stop-off points from among many POIs (Points of Interest). Since there are many POIs in the real world, it is difficult to select appropriate stop-off points unaided. A POI recommendation function may help a user in creating drive plans, but existing recommendation functions offer little variation in plans, lack adaptation to the user's preference and can't consider if a whole drive plan is appropriate for the user. Our approach is to apply context-aware applications to telematics to address these issues. In this paper, we propose an architecture to implement telematics services as a context-aware application. We use a reactive context-aware framework that outputs recommendation contents according to inputs such as a preference and a situation of a user. We also implemented a drive planner that simulates travel corresponding to recommended drive plans to predict a feature situation.

Section 2 of the paper explains problems in applying a context-aware application to a telematics service. Section 3 describes the architecture of Ubiquitous Personal Agent (UPA), our reactive context-aware framework. In Section 4, we provide a detailed explanation of Smart Ride Assist @ Navi (@Navi), which is a telematics service implemented with UPA. In Section 5, we evaluate if the combination of a drive planner and UPA is more effective in creating suitable plans than in the case of using only a reactive context-aware framework. We present related work in Section 6.

2 Applying Context-Aware Application to Telematics Service

In creating a drive plan that contains stop-off POIs, existing telematics services force a user to select a point from a catalog of a large number of POIs. To avoid the burden of selecting points, a telematics service should satisfy the following conditions.

– Creating drive plans that considers a user's preference, a user's situation on the drive, a drive route and a time constraint.
– Use interaction with a user as feedback to the system and personalize behavior of the system.

We propose a method to realize a service that satisfies these conditions by applying a context-aware application to a telematics service. We define "context" as follows.

– Context is information to characterize a situation, preference and objective of a user. That information must be abstracted for ease of use for an application. A situation includes spatial information (e.g. location), temporal information, physiological measurements (e.g. emptiness, fatigue) and activity.

A context-aware application is defined as an application that provides information or performs services according to inputs of context [6][11]. Most existing context-aware applications [7][12][5] work reactively and provide information in response to inputs from sensors.

Context-aware telematics service can recommend a POI at which a user should stop in a certain situation at a certain moment. But such a reactive service has the following problems.

– Such a reactive service recommends a POI only with an estimation at a certain moment, even if there are more appropriate POIs in the feature, a service can't recommend them. For example, a service recommended a restaurant at a certain point and a user had lunch at that restaurant. After that, even if there is another restaurant that is more to the user's taste, the service can't recommend it, and so the total satisfaction rating of the drive plan becomes lower.

For solving this problem, there is an approach in which a static user's preference is used to select POIs. But this approach still has a problem.

– Since each POI is recommended according only to a user's preference, dynamic situations of the user, such as hunger and sleepy, are not reflected in a drive plan.

We solve these problems with a prediction of a user's feature context. In our proposed architecture, a drive planner simulates an itinerary and provides a predicted car's location to a reactive context-aware framework, and recommended POIs are passed from the framework to the planner. The planner creates a drive plan by combining recommended POIs. A drive planner predicts a feature context and evaluates a comprehensive satisfaction rating of drive plans to cover the shortcomings of a reactive context-aware framework.

3 Ubiquitous Personal Agent

We have implemented our reactive context-aware framework, Ubiquitous Personal Agent (UPA) [9]. UPA has the following features.

– UPA abstracts data acquired from a ubiquitous sensor network with using context recognition rules.
– Service selection rules evaluate user's context and output appropriate contents and services for the user in that context.

3.1 Entity, Context and Module

UPA works as follows:

1. Data from various sensor devices are stored in UPA. UPA handle these data with a structure called "Entity". Entity includes data such as a current date and a user's location.
2. "Context" is derived from entity by context recognition rules. Context includes data such as a user's preference and neighboring POIs. Examples of context recognition rules are presented in Section 4.
3. "Module" is derived from context by service selection rules. Module represents services that should be provided to the user in a certain context. Examples of service selection rules and modules are presented in Section 4.

UPA handles entity, context and module with a structure called 'Node'. Node is an architectural component in UPA and it contains a common data set for handling entity, context and module. Rules in UPA fire when a specific combination of nodes is stored in UPA, and output other nodes. Each node and each rule has a weight that represents its importance. A weight of the output node is calculated with weights of rules and nodes that derive the output node.

3.2 Feedback

Each node has information from which rule it is derived. So if a context or a module is considered to be inadequate by a user, UPA can reduce weights of rules derived for that context or module. It works as a feedback to the system to reduce a probability of providing inadequate information to the user.

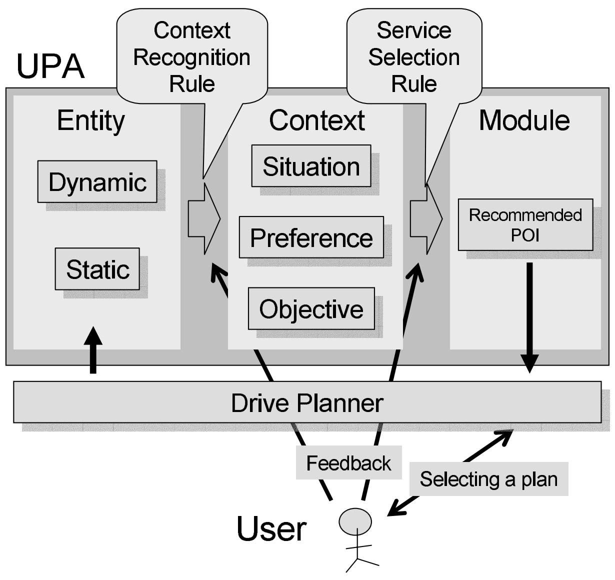

Fig. 1. @Navi Architecture

4 Creating Drive Plans with UPA

In this section, we describe our telematics service @Navi that creates drive plans for a leisure trip with UPA. @Navi recognizes a user's preference and context with rules in UPA, and recommends stop-off POIs. @Navi has a drive planner that covers the shortcomings of reactive action of UPA, obtaining the measure of recommended POIs through the viewpoint that the whole drive plan is appropriate for the user.

4.1 Architecture of @Navi

Fig. 1 shows the outline of the architecture of @ Navi. A drive planner provides time and location to UPA as entities. UPA derives contexts and modules from entities. A drive planner receives modules that contain information about recommended POIs and creates several plans with combinations of these POIs. The user can select and modify the proposed drive plans. Interactions with the user in selecting plans and making changes to stop-off POIs are used as feedback to rules in UPA.

4.2 Using @Navi

In this section, we show how @Navi is used to create a drive plan.

1. Inputting user's preference with user cards
   User cards are data that contains the user's preference and an objective of a drive. For example, user cards have a description like 'A taste for Japanese

food", "Outdoors type", "Sightseeing" and "Travel deluxe". Each card is related to the context node in UPA, and when the card is selected by the user, related context nodes are set to UPA before a drive planner starts creating drive plans.

2. Showing drive plans
   After setting user cards, the starting point, the destination, the starting time and the arrival time, @Navi create several plans in consideration of these constraints. Created drive plans are shown on GUI. The user can select a drive plan and see the detail of each POI that includes the outline of that POI and the estimated context of when the user reaches that POI.
3. Making changes to plans
   The user can select a plan and make changes to POIs in a drive plan. A POI details screen has a button to insert or change the POI. When the user inserts or modifies POIs, @Navi proposes alternative POIs in the drive plan graph. The details of the drive plan graph are described below. Interaction between user and @Navi to modify plans is used as feedback to the system. If the user deletes or changes a POI, weights of rules deriving that POI decrease. If the use selects a POI as an alternative, weights of rules deriving that POI and POIs in the same category increase.

Designing UPA nodes and rules for @Navi. In this section, we describe the details of nodes and rules for @Navi.

Designing nodes. Nodes are classified into entity, context and module. Table 1 shows types of nodes used in @Navi. A node is defined by a name and types of attributes. These nodes have a same weight when they are entered into UPA. All POIs are input as entities to UPA, and these contain genre information as an attribute. By genre information, POIs are classified into 300 categories.

Designing rules. Rules are classified into context recognition rules and service selection rules.

– Context recognition rule
  A context recognition rule derives context such as neighboring POI, time zone, season and hunger. A neighboring POI is derived from entities of a current location, information of POI and threshold distance. Time zone, season and hunger are derived from entities of a current time.
– Service selection rule
  A service selection rule recommends POI according to the user's context. Genre information described in attributes of POI is used for POI recommendation. Service selection rules are created from the mapping table from the user's context to the POI's genre.

Example of a context recognition rule. Fig. 2 shows an example of a context recognition rule. UPA provides a feature to define enumeration rules. Since each enumerated rule can be assigned a different weight, enumeration rules

Table 1. Nodes Used in @Navi

Entity	
Dynamic	Current Location(Coordinate) Current Time(Date) Threshold for Neighboring POI (NearbyThreshold) Stop-off POI(VisitingPoint)
Static	POI(POI)
Context	
Situation (Spatial)	Neighboring POI(NearbyPOI)
Situation (Temporal)	Time Zone(Time), Season(TimeOfYear)

Situation (Physiological)	Hunger(Hungry), Fatigue(Thirsty)
	Sleepy(Sleepy), Urinary(Urinary)

Situation (Activity)	Searching POI(SearchedPOI) Visiting POI(VisitingPoint)

Preference	Gender(UserGender)
	Generation(UserGeneration)
	Passenger(Passenger)
	Estimated cost(Budget)
	Food Preference(FoodPrefrence)
	Outdoors or Indoors type(ActivityType)

Objective	Objective(Purpose)
Module	
	Recommended POI(RecommendPOI)

can define time-varying weights by defining multiple rules corresponding to hours of a current time entity and genre-varying weight with rules corresponding to the genre of a POI entity.

- Defining enumeration rule
  An enumeration rule is defined with an identifier and a set of values. Each expanded rule can has a different weight. In Fig. 2, the enumeration rule has an identifier 'Hour' and a set of values from 0 to 23. The expanded rules that have a value 7, 12, and 19 have a weight of 2.5, rules that have a value 6, 8, 11, 13, 15, 18 and 20 have a weight of 1.5, rules that have a value 9, 10, 14, 16, 17 have a weight of 0.5, and other rules have a weight of 0.1. These rules emulated the hunger context of each hour.
- Defining fire condition
  The fire condition is defined with output context name, input entity name and condition description. In Fig. 2, huger context is derived from Date and VisitingPoint entity.

Example of a service selection rule. Fig. 3 shows an example of a service selection rule. By this rule, RecommendPOI module is derived from user's food

```
rule DeriveHungryState {
 // Defining enumeration rule
 Hour = [0 1 2 3 4 5 6 7 8 9
         10 11 12 13 14 15 16
         17 18 19 20 21 22 23];
 // Defining weights
 weights[Hour hour] {
   [*] = 0.1;
   [9,10,14,16,17] = 0.5 ;
   [6,8,11,13,15,18,20] = 1.5 ;
   [7,12,19] = 2.5; }
 // Defining fire condition
 Hungry hungry
 (Date[maxCardinality=1] date,
  VisitingPoint[maxCardinality=1]
   visitingPoint*[large_class==
    "eating"]) {
      --- snip ---
```

Fig. 2. Example of a Context Recognition Rule

preference and neighboring POIs. RecommendPOI module contains information about the name and the ID of the recommended POI.

4.3 Drive Planner

A drive planner simulates an itinerary of drive plans from the starting point to the destination, and provides predicted situation such as a location and time to UPA. UPA provides recommended POIs to a drive planner, and a drive planner sorts out an appropriate POI and creates a drive plan within constraints of an arrival time and a geographical route. To create variations in plans, a drive planner uses a data structure called a drive plan graph.

Drive plan graph. A drive plan graph is a data structure to manage drive plans from the starting point to the destination. A drive plan graph has nodes that represent each stop-off POI. A path from the node of the starting point to the node of the destination represents a drive plan. A drive plan graph is created as follows (Fig. 4).

1. Creating an initial drive plan graph that contains only the starting point and the destination, and searching the geographical route.
2. Simulating a drive along the route and providing a location and time to UPA at intervals. UPA recommends POIs to a drive planner with a Recommend-POI module.
3. Connecting n POIs next to the node of the starting point as stop-off POIs. These POIs are selected from recommended POIs ranked with weights of corresponding UPA modules.
4. Searching routes from each stop-off POI to the destination.

```
rule DeriveRecommendPOI {
      --- snip ---
 weights[FoodGenre foodGenre,
         POIClass poiClass] {
  [* *] = 0.1 ;
      --- snip ---
  ["Japanese" "Curry", "Ramen"]
    = 0.5; }
 RecommendPOI recommendPoi
  (NearbyPOI nearbyPoi,
   FoodPreference foodPref) {
   if(nearbyPoi.class  == poiClass &&
    foodPref.type == foodGenre ) {
    recommendPoi.id  = nearbyPoi.id;
    recommendPoi.name  = nearbyPoi.name;
       --- snip ---
```

Fig. 3. Example of a Service Selection Rule

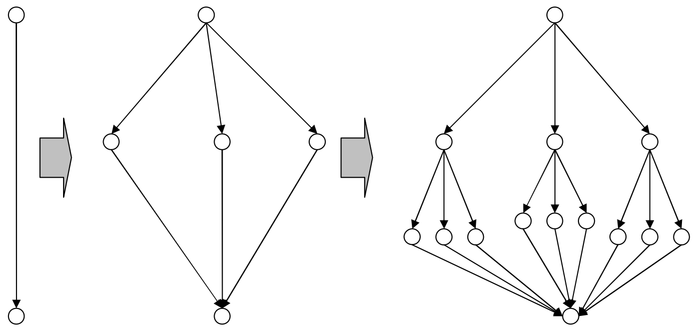

Fig. 4. Creating a Drive Plan Graph

5. Repeating 2 - 4 as long as a user can reach the destination before the arrival time with the recommended drive plan. We assume that the user stays for an hour at each POI.

Drive plans are selected from the drive plan graph as follows.

1. Adding all weights of POIs for each path from the starting point to the destination. Weights of POIs correspond to weights of modules recommending that POI. Total of weights is used as the evaluated value for the drive plan.
2. Selecting the drive plan that has the highest value.
3. To minimize the inclusion of duplicate POIs between plans, if there are POIs in a drive plan graph that are included in the selected plan, weights of these POIs are reduced to half.
4. Repeating the above operations until a predetermined number of plans is selected.

5 Evaluation

In this section, we evaluate if a drive planner can create better plans for a user than in case of only using a reactive context-aware framework. Also, we evaluate if the feedback from the user's operations can improve created plans.

5.1 Creating Better Plans by Using a Planner

To evaluate if the proposed architecture using a combination of a drive planner and a reactive context-aware framework can create better plans than in case of only using a reactive context-aware framework, we experimented with two methods of creating a drive plan with UPA, and checked an evaluated value of each method.

– Planning method: Creating drive plans with a drive planner and UPA. UPA uses the user's feature information predicted by a drive planner. (the method proposed in this paper)
– Reactive method: Stopping at POIs recommended by UPA one after another. UPA uses only the user's current location.

In the reactive method, the system inputs a current location and a current time as an entity to UPA, and stops at the POI that has the highest weight.

An evaluated value of the drive plan is calculated by adding the weight value of each stop-off POI. The weight value of POI is the same as that of its corresponding module. Each method creates a plan that can reach the destination before the arrival time. For each method, we created drive plans with the settings in Table 2 with 4929 POIs.

Fig. 5 shows evaluated values of plans. In most plans, the planning method got a higher value than the reactive method. In drive plans 2 and 8, the reactive method got a higher value, but it was attributable to the larger number of POIs contained in a plan. Fig. 6 shows an average calculated by dividing an evaluated value by the number of POIs in a plan. This figure shows the planning method got a higher value even in plans 2 and 8. We evaluated another 20 plans. Table 3 shows the average value of each method with 30 trials. Table 4 shows P values in a Wilcoxon matched-pairs signed-rank test. P values show that there are statistically significant differences between two methods with significant level $\alpha = 0.01$. So the result shows that the planning method can create the plan that has a higher value and is suitable for the user.

5.2 Improvement Through Feedback

@Navi uses interactions with user as feedback to UPA. In this section, we evaluate that @Navi can make an improvement with feedback and create more appropriate plans. We use the user model to simulate the user's behavior that changes according to preference and context. The user model has a mapping table from user's context and genre of POI to the satisfaction rating of user, and evaluates plans recommended from @Navi. The user model interacts with UI of @Navi,

Table 2. Settings of Evaluated Drive Plans

	Date	Starting - Destination	Time	User Card
1	2004/8/4	Sontoku Memorial	12:00	Alone, Japanese food, Refresh, Deluxe
		Ashinoko Lake	18:00	
2	2004/10/10	Hakone Museum	9:00	Asian food, Sightseeing
		Kanakawa Suisan	15:00	
3	2004/11/18	Sontoku Memorial	12:00	Alone, Japanese food, Refresh, Deluxe
		Ashinoko Lake	18:00	
4	2004/11/18	Odawara Castle Park	9:00	Family vacations, Experience
		Ashinoko Park	15:00	
5	2004/12/20	Tohi Castle	13:00	Reflesh, With friends, Outdoors type
		Hakuundo Teahouse	20:00	
6	2005/1/1	Gotenba Sports Guarden	10:00	Shopping, Italian food, Chinese food
		Kamakura Station	17:00	
7	2005/4/5	Hakone Kougen Hotel	9:00	With lover, Youth, Indoors type
		Dynacity West	14:00	
8	2005/6/21	Kodomo Playground	12:00	Female, Sight scenery, With children
		3D Dinosaur World	18:00	
9	2005/7/7	Hakone Sekisho	10:00	Economy, Sightseeing, Youth
		Katufukuzi Temple	16:00	
10	2005/8/10	Ohkurayama Memorial	14:00	Adult, Chinese food
		Odawarazyou Muse	20:00	

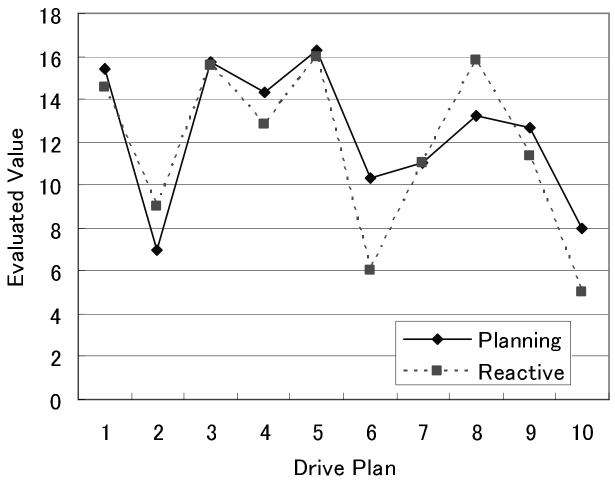

Fig. 5. Evaluated Values of Plans

by selecting a plan and modifying POIs according to the evaluated value of each POI. Table 5 shows a setting of the evaluation. In this evaluation, we set the user card of 'A taste for Japanese food', 'male', and 'adult'. For the user model, we set the preference of 'A taste for Japanese food', 'dislike foreign food' and

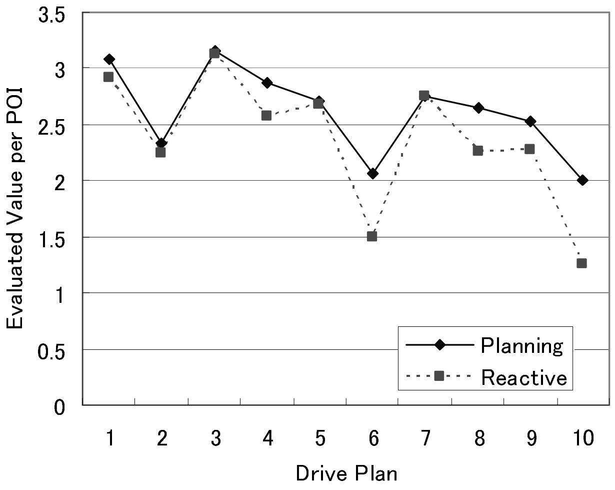

Fig. 6. Evaluated Values of POIs

Table 3. Average of the Evaluated Values

	Value of Plans	Value of POIs
Plannning	11.54	2.49
Reactive	10.27	2.25

Table 4. P values in a Wilcoxon matched-pairs signed-rank test

	Value of Plans	Value of POIs
P value	0.00028	0.00001

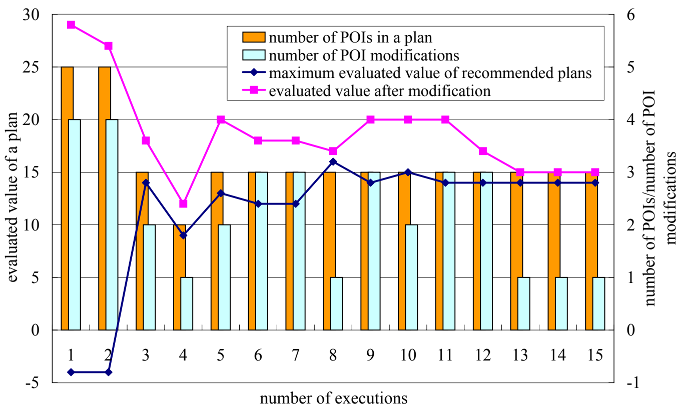

Fig. 7. Evaluated Value and Number of Modifications

Table 5. Settings of Evaluation of Feedback

User Card	Japanese food, Male, Adult
Date	2004/2/1
Starting	Odawarazyo Castle 11:00
Destination	Hakone Prince Hotel 18:00
User Model	A taste for Japanese food, Dislike foreign food and hot spring

'dislike hot springs'. Since some preferences are more detailed descriptions than the user card, we can check if the system can suit the user's detailed preference by using feedback. Fig. 7 shows a result of 15 trial runs of creating plans and simulating a drive. The horizontal axis shows the number of executions and the vertical axis shows the evaluated value of a plan and the number of POI modifications. This result shows the evaluated value is improved by feedback and the number of modifications decreases. So by using feedback, @Navi can adapt to the user model.

6 Related Works

For creating a context-aware application in the ubiquitous environment, pervasive middleware is proposed [4]. Context Toolkit [7] is one of these middleware that provides the widget for hiding the complexity of sensors and providing a common interface to the application and the interpreter for abstraction of context. The developer should implement an algorithm in the interpreter to abstract context. In contrast, UPA provides the framework of rule-based abstraction. SOCAM [12] is another middleware that provides a context model based on ontology and the context interpreter that abstracts the context. The context interpreter has a rule-based reasoning engine like UPA, but rules in SOCAM do not provide a function to handle the importance of rules that is realized with a weight of rule in UPA. SOCAM is used with the telematics service based on OSGi [5]. TRM [3] is another middleware for developing a context-aware telematics service. TRM emphasizes finding and gathering information in a wireless network. TRM provides a language called iQL [10] to abstract data. iQL also describes a function that outputs a abstracted data for a specific input in the same manner as the context recognition rules in UPA, but that function can't handle the feedback from the user. The importance of precision of the user's context has been discussed regarding the context-aware application of wireless mobile devices [8]. The application to support travelers is considered for instance, to explain the advantage of use of context precision to improve the response time. In this paper, we also discussed about the context precision, but our main purpose is to create a drive plan that can adapt to the user.

7 Conclusions

We developed a drive plan recommendation system @Navi with the context-aware application framework UPA. We evaluated that a combination of a reactive context-aware application framework and a drive planner can create appropriate drive plans by using the predicted user's situation. We intend to implement automatic generation of rules based on integration of many users' activities.

References

1. G-book http://g-book.com/pc/.
2. internavi premium club http://premium-club.jp/pr/.
3. C. Bisdikian, I. Boamah, P. Castro, A. Misra, J. Rubas, N. Villoutreix, D. Yeh. Intelligent pervasive middleware for context-based and localized telematics services. *Proceedings of the second international workshop on Mobile commerce, ACM Press*, pages 15–24, 2002.
4. D. Saha, A. Mukherjee. Pervasive computing: A paradigm for the 21st century. *IEEE Computer, IEEE Computer Society Press*, pages 25–31, 2003.
5. Daqing Zhang, Xiaohang Wang, et al. Osgi based service infrastructure for context aware automotive telematics. *IEEE Vehicular Technology Conference (VTC Spring 2004)*, 2004.
6. Dey A.K., Abowd G.D. Toward a better understanding of context and context-awareness. *GVU Technical Report GIT-GVU-99-22, College of Computing, Georgia Institute of Technology*, 1999.
7. Dey A.K., Abowd G.D. The context toolkit: Aiding the development of context-aware applications. *Workshop on Software Engineering for Wearable and Pervasive Computing , Limerick, Ireland*, June 2000.
8. Brown P.J. Jones G.J.F. Exploiting contextual change in context-aware retrieval. *Proceedings of the 17th ACM Symposium on Applied Computing (SAC 2002), Madrid, ACM Press, New York*, pages 650–656, 2002.
9. Masanori Hattori, Kenta Cho, Akihiko Ohsuga, Masao Isshiki, Shinichi Honiden. Context-aware agent platform in ubiquitous environments and its verification tests. *First IEEE International Conference on Pervasive Computing and Communications (PerCom'03)*, 2003.
10. Norman H. Cohen, Hui Lei, Paul Castro, John S. Davis II, Apratim Purakayastha. Composing pervasive data using iql. *Fourth IEEE Workshop on Mobile Computing Systems and Applications*, 2002.
11. Schilit B.N., Adams N.I. and Want R. Context-aware computing applications. *Proceedings of the Workshop on Mobile Computing Systems and Applications. IEEE Computer Society, Santa Cruz, CA*, pages 85–90, 1994.
12. Tao Gu, H. K. Pung, et al. A middleware for context aware mobile services. *IEEE Vehicular Technology Conference (VTC Spring 2004)*, 2004.