Inconsistency Detection and Resolution for Context-Aware Middleware Support

Chang Xu
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong, China
changxu@cs.ust.hk

S.C. Cheung
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong, China
scc@cs.ust.hk

ABSTRACT
Context-awareness is a key feature of pervasive computing whose environments keep evolving. The support of context-awareness requires comprehensive management including detection and resolution of context inconsistency, which occurs naturally in pervasive computing. In this paper we present a framework for realizing dynamic context consistency management. The framework supports inconsistency detection based on a semantic matching and inconsistency triggering model, and inconsistency resolution with proactive actions to context sources. We further present an implementation based on the *Cabot* middleware. The feasibility of the framework and its performance are evaluated through a case study and a simulated experiment, respectively.

Categories and Subject Descriptors
D.2.1 [**Software Engineering**]: Requirements/Specifications – *Methodologies*

General Terms
Algorithms, Design, Management, Performance

Keywords
Pervasive Computing, Context Modeling, Context Consistency Management, Semantic Matching, Proactive Repairing

1. INTRODUCTION
Pervasive computing environments encompass a spectrum of computation and communication devices that seamlessly augment human thoughts and activities [21]. Applications in this type of environments are often context-aware, using various kinds of context such as location and time to adapt to the evolving environments and provide smarter services. For example, a mobile phone would vibrate rather than beep in a concert if the system knows the user's location. Pervasive computing applications need to be context-aware in order to respond quickly to their dynamic computing environments. The growing demand of context-awareness poses an impending requirement on *context consistency* management.

ESEC-FSE'05, September 5–9, 2005, Lisbon, Portugal.

In pervasive computing, the *context* of a computation task refers to the circumstances or situation in which the task takes place (e.g., user's current location and activity). *Context consistency* is maintained when there is no contradiction in a computation task's context; otherwise *context inconsistency* is said to occur. To understand the meaning of context contradiction, let us consider a scenario from the healthcare industry:

Peter is a doctor working for Hope Hospital. He carries a Personal Digital Assistant (*PDA*) as his agent for arranging daily activities. Many kinds of context, such as the environment in which Peter is working, the room in which Peter is located, and the condition of a patient being taken care of by Peter, may affect the agent's suggestion for Peter's next activity. Suppose that at some time the agent acquired three context pieces from different sources:

(1) Peter is in the operating theatre (user location);
(2) An operation is being performed in Room 3504 (room status);
(3) Peter is looking up medical resources (user activity).

From (1) and (2), the agent would probably conclude that Peter is occupied with an operation based on its pre-obtained information that Room 3504 is the operating theatre. However, from (3) the agent might draw another conclusion that Peter is not attending an emergency and therefore able to help patient Michael immediately if he becomes unconscious at any moment. The two opposite assessments reflect the contradiction in the current context, i.e., a conflicting understanding of the surrounding environment. Thus the agent might have difficulty in deciding whether to guide Peter to Michael to check his condition or forward this request to another doctor. As a result, the agent might fail to function correctly.

There are a number of reasons why context inconsistency occurs. In the above scenario, it could be inaccurate location detection (e.g., Peter is passing by instead of actually staying in the operating theatre) or incorrect activity reasoning (e.g., Peter is walking around his office desk on which are some medical resources instead of looking up them at that moment). However, regardless of which reason, the agent can hardly detect and resolve such inconsistencies by itself due to the lack of adequate reasoning capabilities, global situation assessment and effective repairing actions, which typically require considerable computing resources that are not available for portable pervasive computing agents.

An unfortunate observation is that context inconsistency is commonly found in real-life systems. *ActiveCampus* is a real-life example presented in [8]. When context inconsistency occurs due to stale data, *ActiveCampus* is unable to correctly estimate a person's location, which could affect the normal functioning of some services. Our research shows that the occurrence of context inconsistency stems from the natural imperfectness of context:

- **Highly dynamic environments make context easily obsolete** [11]**:** For example, the location context of a fast-moving subject (e.g., a doctor running to an emergency) is prone to errors.
- **Context can be offered by heterogeneous sources under different standards:** Various sensing technologies and standards may lead to semantically contradicting context (e.g., "inside the room" vs. "near the door but outside the room").
- **Context reasoning may introduce inaccurate information due to computing-resource limitation:** The requirement of real-time response (i.e., time limitation) may result in partial consideration of available context in inferring high-level context such as user activity.
- **Network disconnection or failures lead to incomplete context** [11]**:** The mobility of pervasive computing increases the chances of context loss (e.g., "Peter and Michael enter the operating theatre" vs. "only Peter enters the operating theatre").

The natural existence of these imperfect sources makes context inconsistency a common phenomenon. It is difficult to guarantee the correctness, integrity and non-redundancy of context in pervasive computing. However, this problem has not been explicitly addressed by existing context-aware systems (e.g., *Context Toolkit* [6], *EgoSpace* [13], *Gaia* [21] and *Aura* [25]). To overcome this, we have identified two key issues:

- **Inconsistency detection:** Context inconsistency is a semantic phenomenon rather than a syntactic one, whose detection requires non-trivial reasoning work. For example, a context piece "free / not in an emergency" may contradict with "performing an operation", while it can coexist with "looking up medical resources". Usually, the detection is based on common sense and user-specified rules.
- **Inconsistency resolution:** Context evolution in pervasive computing is dynamic and fast, which requires an automatic inconsistency resolution mechanism. Moreover, simple repairing on current context is inadequate for maintaining a stable running environment for applications. Proactive control or feedback to context sources is required to prevent future inconsistencies.

To the best of our knowledge in the existing work on pervasive computing, a systematic study of these two issues has not been conducted. Although it could be argued that they are similar to the evidence aggregation problem [24] from the artificial intelligence (*AI*) discipline, the similarity only lies in that both of them relate to information inconsistency. The causes of inconsistency and the corresponding challenges in resolving it actually differ a lot (please refer to Section 2). Moreover, this paper aims at proposing a consistency management framework using software engineering methodology rather than working on sophisticated inconsistency detection algorithms using *AI* techniques.

The remainder of the paper is organized as follows. Sections 2 and 3 introduce related work of recent years and preliminary concepts on context modeling, respectively. Section 4 presents our framework for context consistency management by focusing on complex context and constraints modeling, and inconsistency detection and resolution. Section 5 briefly introduces the implementation of *Cabot* [26] – a middleware that supports context consistency management. This is followed by a case study in Section 6 and a simulated experiment in Section 7. Section 8 discusses the feasibility of adapting existing technologies to our work. The last section concludes the paper.

2. RELATED WORK

Existing studies on context-awareness are mostly concerned with either the frameworks that support context abstraction or the data structures that support context queries. Pioneering work by Schilit et al. [22] proposes using environment servers to manage context. The context model in this work is simple. Schmidt et al. present in [23] a layered processing model in which sensor outputs are transformed into cues that comprise a set of values with certainty measurements. Gray et al. in [7] are concerned with capturing context meta-information that describes features such as representation, quality, source, transformation and actuation. Harter et al. in [9] propose a conceptual context model that is constructed using an entity-relationship based language. Henricksen et al. in [11] comprehensively analyze context by covering temporal characteristics, information imperfection, various representations and high interrelation. These works are mostly concerned with context modeling techniques, while the problem of context inconsistency is not adequately addressed. Advanced issues about inconsistency detection and resolution are rarely discussed.

Some research projects, e.g., *Gaia* [21], *Aura* [25] and *EasyLiving* [3], have been proposed to provide middleware support for pervasive computing. They are mainly concerned about the organization of and the collaboration among pervasive computing devices and services. Other infrastructure projects mostly focus on the context processing, reasoning and programming support. An earlier representative work is *Context Toolkit* presented in [6]. It assists developers by providing abstract components (e.g., context widgets, interpreters and aggregators) that can be connected together to capture and process context data from sensors. *Context Toolkit* falls short in supporting highly-integrated context applications. To overcome this, Griswold et al. in [8] propose to apply a hybrid mediator-observer pattern in the system architecture. Henricksen et al. in [10] present a multi-layer framework which supports both branching and triggering programming models. Ranganathan et al. in [19] discuss how to resolve potential semantic contradictions in context by reasoning based on first-order predicate calculus and Boolean algebra. In [20] they extend the work to reasoning about context uncertainty using *AI* mechanisms like fuzzy logic. These works have tackled several challenges in context processing, reasoning and programming, and conducted preliminary research on context certainty representation and uncertainty reasoning, but inadequate attention has been paid to the repairing of inconsistent context.

Pervasive computing, a relatively new but fast growing discipline, shares many observations and technology with *AI*, active databases and software engineering disciplines. In the *AI* discipline, expert systems have been developed to support intelligent strategy making. Much effort has been made on the evidence aggregation problem so that the systems are able to make reasonable strategies based on contradicting evidences or rules, but the causes of inconsistency are rarely addressed. Composite event detection is an important issue in the active databases discipline for triggering pre-defined actions once desired events are detected. *E-brokerage* [14] and *Amit* [1] are two widely known projects aimed at detecting composite event occurrences or situation changes with complex timing constraints. The difference lies in the facts that the former is based on event instance modeling and the latter on event type modeling. In the software engineering discipline, *CARISMA* [4] is proposed as reflective middleware support for mobile appli-

cations. It focuses on policy conflict resolution that is similar to our work, but it assumes that accurate context information can be collected by probing sensors periodically, which is different from the basis on which our work is built. Nentwich et al. propose a framework for repairing inconsistent *XML* documents based on the *xlinkit* technology [15], which generates interactive repairing options from first order logical formulae that constrain the documents being checked [16]. However, the framework does not support dynamic computing environments. Moreover, repairing documents alone is inadequate for resolving context inconsistency in pervasive computing. Although the above researches help provide similar experience in problem analysis and resolution, their technologies are inadequate for managing context consistency in two aspects:

- **Inconsistency detection:** Complex context constraints (e.g., timing, spatial and data constraints) cannot be directly modeled. For example, the support of generation time, effective time and freshness requirement for context consistency management is beyond the modeling capabilities of existing technologies. In addition, the inconsistency detection algorithm differs due to these new complex constraints.
- **Inconsistency resolution:** Interactive and simple repairing is unsuitable for dynamic and complex pervasive computing environments. The automatic repairing of current inconsistencies and the proactive preventing of future inconsistencies cannot be supported by any of existing technologies.

3. CONTEXT MODELING

Context can be roughly divided into *physical context* and *logical context*. The former is like evidence, recording various events arising in the physical world (e.g., an object's movement and location), while the latter is typically used for situation assessment, only existing in logical models (e.g., a user's intent and mood). Thus, a general data structure is required for context representation. However, we do not adopt a simple representation like name-value pairs or tuple space [13] for the sake of manageability because it often requires multiple tuples to represent a single context piece. On the other hand, neither do we want to list all context characteristics as proposed by Henricksen et al. [11] because of the high management and computation cost.

We define context *ctx* = (*subject*, *predicate*, *object*, *time*, *area*, *certainty*, *freshness*) as a seven-field data structure, where:

- *Subject*, *predicate* and *object* give the content of the context, where *subject* and *object* are related by *predicate* (using simple English sentence structure), e.g., Peter (*subject*) enters (*predicate*) the operating theatre (*object*).
- *Time* and *area* specify the temporal and spatial constraints relevant to the context: *time* represents the time or period in which the context keeps effective (e.g., "10am on Jun 7, 2005" or "from Apr 1 to Jul 1, 2005"); *area* is the place to which the context relates (e.g., "Hope Hospital").
- *Certainty* is a percentage evaluating the probability level of the context (e.g., "90%"), and *freshness* indicates the generation time of the context (e.g., "10 seconds ago").

There are two time-related fields in the structure: *time* and *freshness*. The former is a context's effective time, while the latter specifies a context's generation time. Normally they are different. For example, people relation context "Michael is taken care of by Peter" may have a long effective time (say, two months) but its generation time may have been "two days ago". In pervasive computing, *freshness* is a basic requirement for evaluating context validness because computing environments tend to change fast and the current context may expire quickly. Such consideration is not supported in event detection related studies like [1] and [14].

For the purpose of complex context recognition, two concepts *context instance* and *context pattern* are introduced: a *context instance* is defined by instantiating all fields of *ctx*, while a *context pattern* (or *pattern* for short) is defined by instantiating some of its fields. Each uninstantiated field (if any) is set to *any*, which is a special predefined value. Intuitively, each pattern represents a family of context instances.

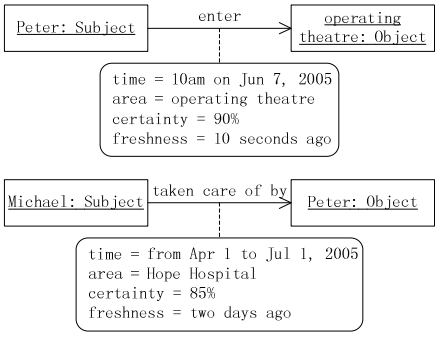

Figure 1. Two context instances

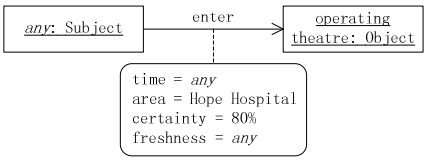

Figure 2. A context pattern

Figure 1 illustrates two context instances in a *UML* object diagram, which represents that: (1) Peter enters the operating theatre, and (2) Michael is taken care of by Peter. Figure 2 illustrates a pattern that represents such context instances as somebody entering the operating theatre.

4. MANAGING CONTEXT CONSISTENCY

A key requirement in context consistency management is the ability to bridge the gap between the context recognized by the middleware and the inconsistency to which the middleware needs to react. This paper aims at bridging the gap by presenting a comprehensive consistency management framework for context in pervasive computing. Three requirements have been identified for this type of computing environments:

- **Semantic reasoning:** Context inconsistency is a semantic phenomenon, which requires necessary reasoning for inconsistency detection.
- **Automatic resolution:** Context evolution is dynamic and fast, which requires an automatic resolution mechanism for any detected inconsistency.
- **Feedback control:** Repairing on current context is inadequate, which requires feedback to context sources to prevent future inconsistencies.

4.1 Model Complex Context and Constraints

Let us first take a look at an example of complex context:

A doctor enters the operating theatre, where an operation is going to be performed in ten minutes on a patient, who now looks a little nervous.

This example contains several context pieces, including physical ones (e.g., a doctor's location) and logical ones (e.g., a patient's state of mind), and some constraints, including timing constraints (e.g., "an operation will be performed *in ten minutes*"), spatial constraints (e.g., "the doctor and the patient are *in the same room*") and data constraints (e.g., "*the person* entering the room *is a doctor*"). To model such complex context, we begin with basic blocks (e.g., context instances and patterns in Section 3) and use operations (e.g., *context matching*) to connect them together.

4.1.1 Semantic Context Matching

A fundamental operation, *context matching*, is studied below. *Context matching* is a process of checking whether a context instance and a pattern match or not. Unlike other models, our context matching connects context instances and patterns by semantics. Its goal is to integrate basic reasoning into the underlying context model.

There are two usages of context matching: (1) given a context instance, search all matched patterns (pat_mat); (2) given a pattern, search all matched context instances (ins_mat):

$$\text{pat\_mat}(ins, rules) = \{pat \in Patterns \mid \forall field.\text{match}(rules.field, ins.field, pat.field)\}$$

$$\text{ins\_mat}(pat, rules) = \{ins \in Instances \mid \forall field.\text{match}(rules.field, ins.field, pat.field)\}$$

The match function is a kernel process of evaluating whether a given field of a context instance matches its counterpart of a pattern under some *unification rule*. The notation of *unification rules* is based on concept semantic relationships. Let $E(c)$ denote the element set represented by concept c. Any two concepts c_1 and c_2 are subject to one of five semantic relationships [27]:

- *equivalent*: if $E(c_1) = E(c_2)$;
- *subsumed*: if $E(c_1) \subset E(c_2)$;
- *including*: if $E(c_1) \supset E(c_2)$;
- *disjoint*: if $E(c_1) \cap E(c_2) = \phi$;
- *intersecting*: otherwise.

Based on the above five semantic relationships, unification rules express the conditions under which a given context instance and a pattern can be matched. A matching is recognized if each field (except *time*) value v_1 in context instance *ins* is unifiable with its counterpart v_2 in pattern *pat* as follows:

If v_2 = *any*, or v_1 and v_2 satisfy one of six conditions: (1) *identical condition* ($v_1 = v_2$), (2) *equivalent condition* (v_1 and v_2 are equivalent), (3) *plug-in condition* (v_1 and v_2 are equivalent or subsumed), (4) *covering condition* (v_1 and v_2 are equivalent or including), (5) *overlapping condition* (v_1 and v_2 have a non-disjoint relationship), or (6) *unrelated condition* (v_1 and v_2 are disjoint), then v_1 is unifiable with v_2; Otherwise, v_1 is not unifiable with v_2.

Time is a special field following different unification rules including conditions like *close to*, *before*, *after*, *within* and *covering*. These all have intuitive interpretations.

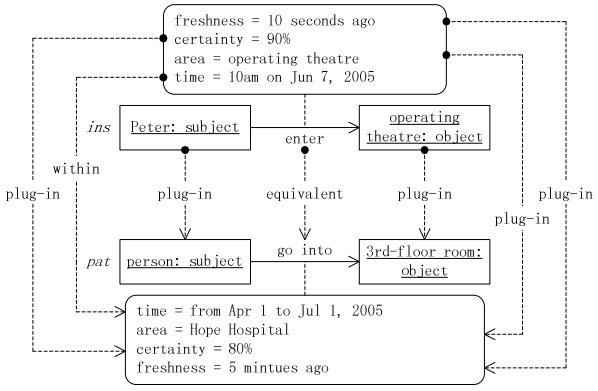

Figure 3. A context matching example

Different fields in a pattern can apply different conditions. Figure 3 illustrates an example, which shows that context instance *ins*, "Peter enters the operating theatre", matches pattern *pat*, "A person goes into a 3rd-floor room". Note that the *certainty* field in *pat* has an "at least" interpretation. As such, *certainty* "90%" in *ins* is unifiable with *certainty* "80%" in *pat* under the plug-in condition. The same interpretation applies to the *freshness* field.

The above example assumes the following concept semantic relationships (which can be inferred from an ontology database that is maintained by the system administrator):

- match("plug-in", "Peter", "person") = true
- match("equivalent", "enter", "go into") = true
- match("plug-in", "operating theatre", "3rd-floor room") = true

Context matching relates context instances and patterns under semantic interpretations, supporting higher expressiveness in context queries than simple byte-by-byte comparisons. As such, it is also known as semantic context matching. In context inconsistency detection (see Section 4.2), automatic reasoning can be supported by semantic context matching.

4.1.2 Complex Context and Constraints

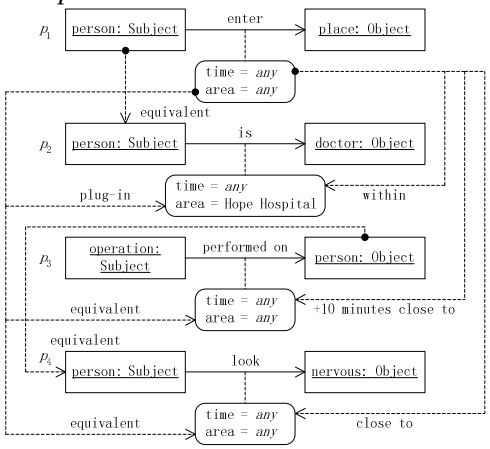

Figure 4. The complex context example

Complex context *ccx* is defined as a group of patterns *patterns* = {pat_1, pat_2, … pat_m} with a group of constraints *constraints* = {cns_1, cns_2, …, cns_n}. Constraints are used to express the relationships between these patterns. They are enforced at runtime. Each constraint takes the form of (*rule*, *j*, $field_j$, *k*, $field_k$), meaning that

if there are two context instances matched for pat_j and pat_k respectively, their values in fields $field_j$ and $field_k$, respectively, should satisfy the given *rule* (unification rule). To make the whole complex context *ccx* assessed to be the current situation, there must be a group of context instances matching each pattern in *ccx* respectively, and these context instances must also satisfy all *ccx*'s constraints.

Figure 4 illustrates the complex context example discussed at the beginning of Section 4.1. It consists of four patterns and eight constraints between them (including three timing constraints, three spatial constraints and two data constraints), represented by dashed lines. We explain three of them for illustration:

- **Timing constraint ("+10 min close to", 1, "time", 3, "time"):** *The time* when a person enters such a place *is 10 minutes before* an operation is performed there.
- **Spatial constraint ("equivalent", 1, "area", 3, "area")**: A person enters *a place where* an operation is going to be performed in ten minutes.
- **Data constraint ("equivalent", 1, "subject", 2, "subject")**: *The person who* enters some place *is* a doctor.

The enforcement of constraints over the "tables" derived from context matching is similar to the *equi-join* in relational databases [18]. Each "table" contains matched context instances for each corresponding pattern, and the join "columns" are specified by constraints. The difference is that "columns" are related by semantics, in particular when we use the equivalent condition which connects two field values of similar meaning (e.g., "enter" and "go into"). This kind of join is called *semantic-join*.

The semantic matching and join used in our model is a major difference from other models. An advantage is that it simplifies the task of describing general context inconsistency (e.g., context "*some person* is performing two *unrelated jobs* at the same time" is considered inconsistent).

4.2 Detect and Resolve Context Inconsistency

We regard context inconsistency as a special kind of complex context, in which situation assessment is subject to inherent contradiction. Based on the previous model preparation, this subsection introduces inconsistency triggering which provides an effective mechanism for inconsistency detection and resolution. Our model of inconsistency triggers is adapted from the Event-Condition-Action (*ECA*) triggers in active database systems [18]. We define an inconsistency trigger as *tgr* = (*event*, *condition*, *action*):

- *Event* is a context-related change that activates the trigger. It specifies a complex context description *ccx* that describes our interested situation. The change occurs when *ccx* is assessed to be the current situation.
- *Condition* is a context-related query that is run when the trigger is activated. It includes a group of patterns that represent a series of tests. Each pattern should match at least one context instance in the context repository such that the whole condition is satisfied.
- *Action* is a procedure that is executed when the trigger is activated and its condition is satisfied.

Figure 5 illustrates how to use inconsistency triggering to describe the problematic situation discussed in Section 1 (the *Action* part is omitted). Please note that constraints also apply to conditions.

4.2.1 Inconsistency Detection Algorithm

Three context types are identified based on their context nature: *sensed contexts* (e.g., Peter enters the operating theatre) are collected by sensor devices; *domain contexts* (e.g., Peter takes care of Michael / Michelle was born in Jan 1977) are supplied by human operators; and *derived contexts* (e.g., Michael becomes unconscious) are computed by software programs based on existing contexts. Sensed and derived contexts typically change more frequently than domain contexts.

The execution of an inconsistency trigger can be divided into three steps: (1) context detection, (2) condition evaluation, and (3) action execution. Step 1 focuses on the monitoring of incoming time-stamped context events (mainly sensed or derived contexts); Step 2 performs queries against stored history contexts (mainly domain contexts).

Context nature is a factor affecting the execution of inconsistency triggers. The detection buffer (or matching queues, please refer to the following algorithm) size is decided by the freshness requirements of related patterns in Step 1. To save memory, usually only sensed and derived contexts are monitored in this step. These contexts often have a strong freshness requirement, leading to short matching queues.

Compared to event detection, inconsistency detection often needs to consider various types of constraints (e.g., timing, spatial and data constraints). Even for timing constraints, inconsistency detection has to differentiate a context's generation time from its effective time, while event detection only focuses on an event's occurrence time. From the perspective of timing constraints, an event's occurrence time is analogous to a context's generation time. So the context inconsistency detection can subsume event detection.

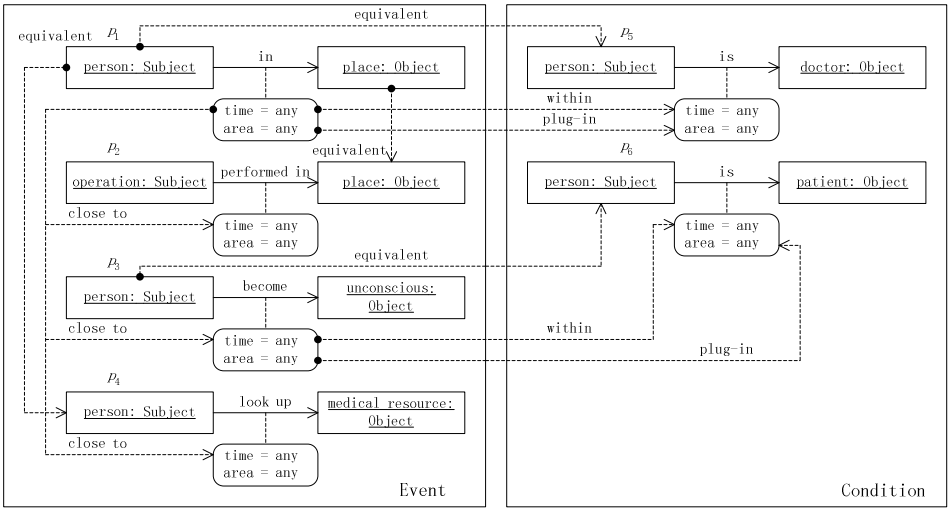

Figure 5. An inconsistency trigger

We give the detection algorithm below:

```
(1) Context Preprocessor Thread (T1):
    wait for any incoming context instance ins
    for each pattern pat matched by ins
      add ins to pat's matching queue pat_que
      if ins is the first element in pat_que
        then create a timer t for pat based on pat's
          freshness requirement and ins's generation
          time
(2) Inconsistency Triggering Thread (T2):
    wait for any new context instance ins in pat's
      matching queue pat_que
    for all other patterns pat1, pat2, ..., and patn
      in pat's owner trigger tgr
      if exists ins1 in pat_que1, ins2 in pat_que2,
        ..., and insn in pat_quen such that tgr's
        constraints on ins, ins1, ins2, ..., and insn
        all satisfied
        then if tgr's conditions also satisfied
          then inconsistency detected
(3) Timer Controller Thread (T3):
    wait for any expired timer t
    remove the first element from t's related
      pattern pat's matching queue pat_que
    if pat_que empty
      then cancel t
      else update t based on pat's freshness
        requirement and the new first element's
        generation time (in pat_que)
```

The algorithm consists of three threads: T_1 performs matching for each incoming context instance, and attaches a copy of it to each matched pattern's matching queue. T_2 monitors all matching queues to see whether there is a group of context instances able to activate a trigger with all its conditions and constraints satisfied. T_1 and T_2 work as a producer-consumer pair of context instances for inconsistency triggering purposes. T_3 manages all running timers and removes expired context instances from their located matching queues when necessary.

According to the classification from *Snoop* [5] for instance consumption, the above algorithm adopts the *continuous* policy [1], i.e., maximizing the use of each context instance according to its relevant freshness requirement (see T_2). The freshness requirement of a pattern specifies the period in which a matched context instance for this pattern stays valid. Under this specification, the algorithm detects all possible inconsistencies among valid captured context instances. In implementation, freshness requirements can be enforced by timers.

The continuous policy for event detection is generally impractical because of its unlimited memory cost but such policy is feasible for our complex context detection. This is because one can control the memory cost by setting a reasonably strong freshness requirement, i.e., a short time period. The maximum memory cost of our implemented framework is below 23MB (including the Java *VM*'s memory cost) under the experimental setting in Section 7.

Another consideration is delay time. New contexts have to be kept in matching queues for a period dependent on relevant patterns' freshness requirements. Fortunately, the delay time is also controllable (fully decided in the design phase by specifying freshness requirement). Users are urged to avoid unreasonably weak freshness requirements in Step 1. Weak freshness requirements should be moved to Step 2, which does not affect the delay time. Another solution is to allow access to these temporary context data (still in matching queues) at the cost of possible inconsistency.

4.2.2 Inconsistency Resolution

The context matching and inconsistency triggering model contributes to inconsistency detection by semantically defining and discovering: (1) the relationships between context instances and patterns, and (2) the relationships between context instances (Figure 6). Once an inconsistency is detected, proper repairing actions need be taken to guarantee the accuracy of context.

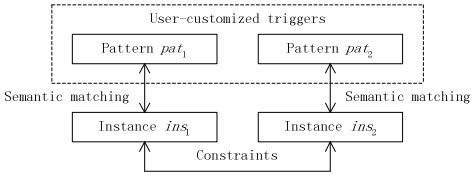

Figure 6. Context matching and inconsistency triggering

Generally, when we detect inconsistency between new and old data stored in an information repository, common actions are to repair the repository based on two types of policies: (1) **Accept policy:** Accept new data into the repository and delete inconsistent old data for inconsistency-resolution; (2) **Reject policy:** Reject new data, and old data remain unchanged. For context, either policy solely focuses on the repairing on the repository, but pays little attention to repair the context sources. As such, the environment may still keep generating inconsistent contexts.

Recently, a substantial amount of work has been made on active systems, which either react automatically to environment changes (reactive systems) or predict changes in their environments (proactive systems) [1]. Concerning inconsistency resolution, the traditional accept/reject policies belong to reactive repairing actions, which work when actual inconsistencies have occurred.

Reactive repairing actions cannot effectively prevent future inconsistencies. To overcome this limitation, we propose a mechanism to support both reactive and proactive repairing actions:

- *Reactive repairing actions* are performed to repair context data in the context repository. This is analogous to the accept/reject policy except that we also support on-demand context update.
- *Proactive repairing actions* are performed to repair context sources, e.g., to control or adjust problematic sensing devices to avoid further occurrences of inconsistent contexts.

Two policies are supported by reactive repairing actions:

1. Static policy:
- Delete pre-specified context instances (e.g., delete the instance matched by pattern pat_1).
  Primitive: `delByPat(patternID)`

2. Dynamic policy:
- Delete most uncertain context instances (e.g., delete the instance with the lowest uncertainty).
  Primitive: `delByUct(LOWEST)`
- Re-query relevant context sources to get a new copy for some context instances.
  Primitive: `uptByPat(patternID, queryTime)`

By default, all context instances kept in the detection buffer will be moved to the context repository automatically when relevant timers expire except for those which have to be deleted according to the static policy. In the dynamic policy, the query time has to be enforced when executing `uptByPat`. Such time enforcement

is usually reasonable and useful as discussed in [12], where in a location re-query example, a one-minute time limit indicates both that the user can afford to wait some time for the query to completed, and that the user desires the location provider to expend a sufficient amount of effort to locate a certain person. Although too long a time limit is unacceptable for the timely resolution of inconsistency, multi-thread technology for parallel processing of inconsistency can alleviate this problem.

Two policies are supported by proactive repairing actions:

1. Active policy:
- Control the lifecycle of a context source (after `delayTime`). Primitive: `srcCtrlByPat(patternID, PAUSE/ RESUME/RESTART/STOP/START, delayTime)`
- Count/get the inconsistency times for/of a context source. Primitive: `incCntByPat/getCntByPat(patternID)`

2. Passive policy:
- Send feedback to a context source and allow it to adjust itself. Primitive: `fdbkByPat(patternID)`

Most sensor devices and software programs support direct control from the middleware on their lifecycles, which makes possible for them to stop generating contexts or restart at a later time when necessary. The passive policy is based on the observation that some advanced context sources can adjust error/uncertainty by changing algorithm parameters (e.g., a location deriving algorithm). A practical example is Microsoft *RADAR* [2] with a 50% uncertainty on its location calculation and a maximum error of 3 meters. The uncertainty can be lower if a greater error is allowed.

The following gives example repairing actions for the hospital scenario we discussed earlier (see Figure 5):

Step 1: repairing context data
```
(1) uptByPat(1,500)
(2) uptByPat(4,500)
(3) int pid = delByUct(LOWEST,{1,4})
```

Step 2: repairing context sources
```
(4) incCntByPat(pid)
(5) int t = getCntByPat(pid)
(6) if (t>2) fdbkByPat(pid)
(7) if (t>5) srcCtrlByPat(pid,RESTART,1000)
(8) if (t>10) srcCtrlByPat(pid,STOP,200)
```

The above code tries to update the context instances matched for patterns p_1 and p_4, and decide which one has the lower uncertainty. For the context source which generates this context instance, its inconsistency counter is increased, and then some action (e.g., feedback sending, restarting or stopping) is taken according to the counter value.

Supporting proactive repairing actions is non-trivial. Different context sources may vary in the support of inconsistency repairing, and a designer may have no knowledge about the context sources involved at runtime. Currently, illegal repairing actions are ignored automatically. For future extension, we are investigating a negotiation-based repairing mechanism which integrates the consideration of learning supported repairing actions at runtime.

5. IMPLEMENTATION

The consistency management framework assumes the availability of an underlying context middleware. We have implemented the framework based on one of our research projects – *Cabot*. *Cabot* is a software infrastructure supporting Context-aware Applications Built on Ontology Technology developed by *JDK* 1.4.2. From *Cabot*'s point of view, a pervasive computing environment is composed of an *application layer*, a *middleware layer* and a *context source layer* (Figure 7).

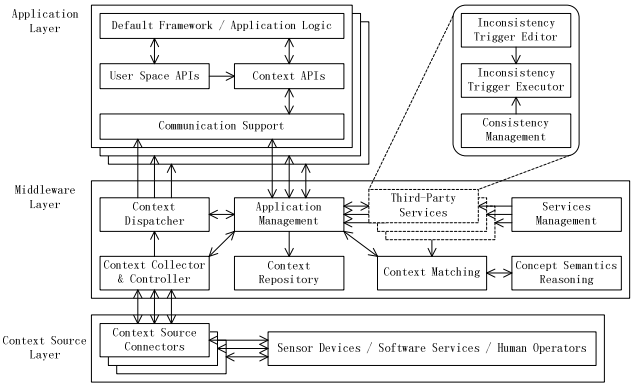

Figure 7. The *Cabot* system architecture

The middleware layer is the kernel part of *Cabot*. It includes five fundamental functionalities: application management, context management, context matching, semantic reasoning and third-party services management. A more detailed introduction to these functionalities can be found in [26].

Our consistency management framework is realized as a third-party service plugged into *Cabot*. When a new context instance arrives, all plug-in services are invoked one by one for context filtering purposes such that management tasks for like context consistency can be achieved. An editor in the framework enables application developers to customize their inconsistency triggers. Repairing actions are also specified in the design phase. Currently, they are implemented through a callback mechanism in terms of user-designed java classes that use the framework primitives (see Section 4.2.2). The framework is responsible for maintaining a consistent context repository. Applications access context of interest via queries or topic subscription.

To support effective context matching and inconsistency detection, the *Cabot* kernel has been rewritten. *Cabot*'s early version was built on the *xlinkit* technology in which computationally expensive checking consumed much processing time. Moreover, semantic-join and complex context detection were not supported in that version. *Cabot*'s current version has increased expressive power for context capture and inconsistency detection. The new detection algorithm is based on the *Amit* technology (see Section 8).

6. CASE STUDY

This section takes an automatic vehicle (*AV*) system based on the Radio Frequency Identification (*RFID*) technology as a case study. *AV* system is one of our ongoing projects on context-awareness with a goal to provide continuous remote control on intelligent vehicles working for humans in an adverse environment (e.g., too dark, dangerous, hot or noisy).

To facilitate the location estimation of vehicles, some reference sites are chosen and installed with *RFID* tags. These tags together with those attached to vehicles are used for tracing each vehicle, routing them to perform designated tasks at different destinations. *AV* system is context-aware in that it controls vehicles based on

the environmental context and each vehicle's condition. Its typical tasks include automatic path selection and collision avoidance.

In practice, certain conditions may introduce incorrect data to *AV* system. For example, a fast moving *RFID* tag attached to a vehicle might be missed by *RFID* antennae (e.g., "detected" vs. "not detected"); overlapped *RFID* tags due to the close proximity of two vehicles could not be always distinguished (e.g., "tag *A* detected" vs. "tag *B* detected"); metal and electromagnetic goods would lead to reduced detection sensitivity (e.g., "no tag active"); and high-level context reasoning services for inferring value-added context (e.g., "vehicle *C* enters area *I*" or "vehicle *D* is stopped in area *II*") might generate incorrect context (e.g., "leave" vs. "enter" or "moving" vs. "stopped").

As a result, context inconsistency naturally occurs in reality and affects the correct functioning of *AV* system. For example, automatic collision avoidance of multiple vehicles would fail if the existence of some *RFID* tags cannot be correctly identified or the current position of a moving vehicle cannot be precisely computed. In practice, multiple sensing technologies (e.g., infrared or ultrasonic) can be used for providing multiple data sources. However, this increases the probability of context redundancy and inconsistency because these technologies use different approaches and standards to compute related context data. *AV* system's strategies may be unexpectedly affected by inconsistent context and possibly generate incorrect control on vehicles.

Suppose that the following context sources have been set up (*S*: sensor devices, *F*: software programs, *H*: human operators (Figure 8):

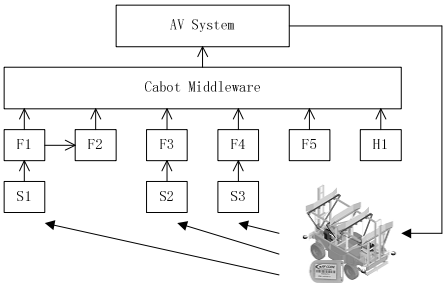

Figure 8. *AV* system and context sources deployment

***S*1:** Four *RFID* detection systems provide signal strength information for the *RFID* tags detected in their sensing ranges.
***S*2:** The ultrasonic sensor installed in each vehicle provides the distance information to its adjacent barriers (e.g., vehicles and goods).
***S*3:** The accelerometer installed in each vehicle provides the tilt and vibration measurements of the vehicle.
***F*1:** The *LANDMARC* algorithm [17] computes the real-time location of each vehicle (from *S*1).
***F*2:** A collision avoidance service reports possible collision when two vehicles are too close (from *F*1).
***F*3:** Another collision avoidance service reports possible collision between a vehicle and its adjacent barriers (from *S*2).
***F*4:** A vehicle status service provides each vehicle's current activity information (e.g., moving, loading or stopped) (from *S*3).
***F*5:** A task management program arranges everyday pre-scheduled goods conveying tasks.
***H*1:** A console interface accepts user's inputs and generates on-the-fly goods conveying tasks.

We consider two major functions of *AV* system:

***SELT*:** According to each vehicle's current location and activity, select the most suitable vehicle (e.g., close to the goods and free of tasks) to carry out a given task.
***CTRL*:** According to the environmental context feedback (e.g., the distance to other vehicles), adjust the controls on each moving vehicle to avoid collisions.

The *SELT* function may be affected by the precision of vehicle location computation, which is not always accurate (e.g., the *LANDMARC* algorithm has an average error of 1 meter under the experimental setting discussed in [17]). The *CTRL* function depends much on the reports from two collision avoidance services, but sometimes they may report inconsistent situations (e.g., "vehicle *C* is close to vehicle *D*" vs. "vehicles *C* and *D* are in different areas"). To alleviate the impact of possible context inconsistency, the following two inconsistency triggers are designed:

***SELT*:** If a vehicle's continuously computed locations vary largely (e.g., more than 2 meters) over a short period of time (e.g., 1 second), a possible location inconsistency occurs. Corresponding repairing actions: update the latest location (enforce query time < 1 second) and delete the old one if they are too different.
***CTRL*:** If two collision reports from *RFID*-based and ultrasonic-based technologies are inconsistent, update the latter, and if they are still different, choose the former and increase the inconsistency counter for the latter. If the counter value reaches 5, restart the relevant ultrasonic sensor, and if the value has been already larger than 10, stop it and write system logs for suggested maintenance (possible damage).

Currently, the project is still under development. The feasibility of our consistency management framework needs further evaluation through practical studies.

7. PERFORMANCE MEASUREMENTS

The goal of performance measurement is to estimate the incoming context rate that *Cabot* can handle. Inspired by the scenario classifications in [1], we have designed four test scenarios:

Standby world: This is an empty scenario that does not define any inconsistency trigger. It gives an upper bound on the performance of *Cabot*'s context processing.
Noisy world: This is a light scenario in which only a low percentage (12%) of the incoming context activates patterns in inconsistency triggers. The inconsistency triggers are not complex, i.e., no conditions or constraints.
Filtered world: This is a filtering scenario in which a high percentage (35%) of the incoming context activates the patterns in inconsistency triggers. However, the conditions of a high percentage (66%) of the activated inconsistency triggers are not satisfied. The inconsistency triggers are relatively complex (i.e., conditions are tested without constraints).
Complex world: This is a heavy scenario in which a quite high percentage (50%) of the incoming context activates the patterns in inconsistency triggers, and the conditions of a high percentage (66%) of these activated inconsistency triggers are satisfied. The inconsistency triggers are very complex (i.e., conditions are tested with constraints).

Experiments were designed for comparing *Cabot*'s performance in the four simulated worlds. They were performed on a Pentium IV 3.20-GHz machine running Windows XP Professional. A context source thread sent 2000 context instances to *Cabot* at 2 instances per second. Example contexts were generated and inconsistency triggers (3) were designed according to the requirement of each scenario (except the standby world). Each inconsistency trigger contains 4 or 6 patterns and 11 constraints (if any). For contrast, the repository contained fixed number (100) of history context instances for condition testing for activated inconsistency triggers. All freshness requirements of patterns in inconsistency triggers were set to 10 seconds. Three parameters were monitored:

(1) Number of incoming context instances, activated triggers (i.e., all event patterns are matched) and triggered inconsistencies (i.e., all conditions are satisfied with constraint enforcement)
(2) Total time (sec), event pattern matching time (sec), condition pattern matching time (sec), constraint enforcement time (sec) and other overhead time (sec)
(3) Processed context instances (per min) and detected inconsistency number (per min)

Table 1. Performance measurement results

	Sta. World	Noi. World	Fil. World	Com. World
Incoming Ctx.	2000	2000	2000	2000
Activated Tgr.	0	39	974	1310
Triggered Inc.	0	69	1632	2250
Total (s)	1.92	23.96	594.59	809.41
Event (s)	0	22.01	65.50	69.33
Condition (s)	0	0	526.04	677.39
Constraint (s)	0	0	0	59.39
Overhead (s)	1.92	1.95	3.05	3.30
Contexts / m	62565	5008.35	201.82	148.26
Inconsistencies / m	0	172.79	164.69	166.79

Table 1 presents the average results of performance measurements of five executions with little difference among them, which show:

(1) *Cabot*'s upper bound was about 62500 context instances per minute. This rate was achieved when none of the incoming context instances takes part in any inconsistency detection.
(2) The lower bound was about 150 context instances per minute. This happened when quite complex inconsistency triggers were activated and evaluated frequently. This kind of case is unlikely to occur in reality.
(3) A relatively high percentage (88.5% for the filtered world and 83.7% for the complex world) of the total time was spent on condition pattern matching. This indicates that the condition evaluation (needs to query all history contexts) is computationally expensive. The reason is that our current implementation cannot utilize mature database technologies that do not support semantic matching and join.

8. DISCUSSION

E-brokerage [14] and *Amit* [1] present two interesting solutions to the problem of event detection. Their solutions based on event modeling are similar to ours in that all three solutions focus on constraint specification and situation detection. *E-brokerage* [14] is based on event instance modeling. Although it is impractical to adopt the continuous policy for event instance consumption because of the lack of controllable constraints on instance freshness requirements (leading to unlimited memory cost), *E-brokerage* [14] utilizes restricted instance relationships (e.g., the time interval between the *i*-th E_1 and E_2 instances) to limit the number of available event instances. However, context detection needs to maximize the use of each context instance within its valid period (specified by freshness requirement) in order to detect any possible inconsistency. The index of an available context instance, which is decided dynamically by its generation time and the relevant pattern's freshness requirement, cannot be modeled directly using restricted instance relationships which are essentially static.

The approach adopted by *Amit* [1] is closer to ours. It is based on event type modeling since any event instance belonging to a relevant event type can participate in the target situation detection. In order to adapt to complex context detection in pervasive computing, *Amit* [1] system's underlying data structures have to be modified to allow for more attributes such as *effective time* and *area* such that complex timing, spatial and data constraints can be modeled. Moreover, the detection algorithm has to be modified to enforce new complex constraints such as freshness requirement. Such adaptation work is non-trivial, and the adaptation result (plus our semantic matching and join for reasoning purposes) is equivalent to our proposed context model.

In inconsistency resolution, *xlinkit* [15] is an excellent tool for *XML* document integrity checking. The major reason why *xlinkit* is not suitable for context consistency management is that it cannot adequately support the regular and frequent detection of information inconsistency. A direct application of *xlinkit* to inconsistency detection in dynamic pervasive computing environments requires repeatedly checking the entire context repository, which is computationally expensive. Our past experience of using it in *Cabot*'s early version exhibited unsatisfactory performance because of the great amount of expensive checking. *Cabot*'s current version outperforms its previous version by 3700%, 450%, 130% and 150% under the four simulated worlds, respectively.

9. CONCLUSIONS AND FUTURE WORK

In this paper, we have studied the natural imperfectness of context in pervasive computing environments, and analyzed the hardness of context consistency management from two aspects: inconsistency detection and resolution. A formal semantic matching and inconsistency triggering model is proposed to capture inconsistent contexts. Then a proactive repairing mechanism is proposed to realize automatic inconsistency repairing. The whole framework has been implemented based on the *Cabot* middleware.

Our framework still has limitations in performance. We are considering more efficient matching algorithms built on mature database technologies. Moreover, the enumeration of all imaginable inconsistencies is somewhat impractical. So we are also working on incremental violation checking techniques for consistency constraints that are more feasible in practice. Other issues such as negotiation-based repairing mechanisms and scalability considerations will be incorporated into our improved framework.

ACKNOWLEDGMENTS

The work is supported by a grant of the Research Grants Council of Hong Kong (Project No. HKUST6167/04E). The authors would also like to thank Michael Liu (lrcomp@cs.ust.hk) for the case study from his led *AV* team.

10. REFERENCES

[1] Asaf Adi, Opher Etzion. Amit – The Situation Manager. *VLDB Journal (13)*, pp. 177-203, 2004.

[2] P. Bahl, V. N. Padmanabhan, A. Balachandran. Enhancements to the RADAR User Location and Tracking System. *Microsoft Research Technical Report*, Feb 2000.

[3] B. Brumitt, B. Meyers, J. Krumm, A. Kern, S. Shafer. EasyLiving: Technologies for Intelligent Environments. *Proceeding of the 2nd International Symposium on Handheld and Ubiquitous Computing (HUC 2000)*, Bristol, England, 2000.

[4] Licia Capra, Wolfgang Emmerich, Cecilia Mascolo. CARISMA: Context-Aware Reflective Middleware System for Mobile Applications. *IEEE Transactions on Software Engineering 29(10)*: pp. 929-944, Oct 2003.

[5] S. Chakravarthy, D. Mishra. Snoop: An Expressive Event Specification Language for Active Databases. *Data Knowl Eng 14.1*: pp. 1–26, 1994.

[6] Anind K. Dey, Gregory D. Abowd, Daniel Salber. A Context-Based Infrastructure for Smart Environments. *Proceedings of the 1st International Workshop on Managing Interactions in Smart Environments*, Dublin, Ireland, Dec 1999.

[7] Philip D. Gray, Daniel Salber. Modeling and Using Sensed Context Information in the Design of Interactive Applications. *Proceedings of the 8th IFIP International Conference on Engineering for Human-Computer Interaction (EHCI 2001)*, Toronto, Canada, May 2001.

[8] William G. Griswold, Robert Boyer, Steven W. Brown, Tan Minh Truong. A Component Architecture for an Extensible, Highly Integrated Context-Aware Computing Infrastructure. *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, Portland, USA, May 2003.

[9] Andy Harter, Andy Hopper, Pete Steggles, Andy Ward, Paul Webster. The Anatomy of a Context-Aware Application. *Mobile Computing and Networking*, pp. 59-68, 1999.

[10] Karen Henricksen, Jadwiga Indulska. A Software Engineering Framework for Context-Aware Pervasive Computing. *Proceedings of the 2nd IEEE Conference on Pervasive Computing and Communications (PerCom 2004)*, Orlando, USA, Mar 2004.

[11] Karen Henricksen, Jadwiga Indulska, Andry Rakotonirainy. Modeling Context Information in Pervasive Computing Systems. *Proceedings of the 1st International Conference on Pervasive Computing*, Zurich, Switzerland, Aug 2002.

[12] Glenn Judd, Peter Steenkiste. Providing Contextual Information to Pervasive Computing Applications. *Proceedings of the 1st IEEE International Conference on Pervasive Computing and Communications (PerCom 2003)*, Dallas, USA, Mar, 2003.

[13] Christine Julien, Gruia-Catalin Roman. Egocentric Context-Aware Programming in Ad Hoc Mobile Environments. *Proceedings of the 10th International Symposium on the Foundations of Software Engineering (FSE 2002)*, Charleston, USA, Nov 2002.

[14] Aloysius K. Mok, Prabhudev Konana, Guangtian Liu, Chan-Gun Lee, Honguk Woo. Specifying Timing Constraints and Composite Events: An Application in the Design of Electronic Brokerages. *IEEE Transactions on Software Engineering 30(12)*: pp. 841-858, Dec 2004.

[15] C. Nentwich, L. Capra, W. Emmerich, A. Finkelstein. xlinkit: A Consistency Checking and Smart Link Generation Service. *ACM Transactions on Internet Technology 2(2)*: pp. 151-185, May 2002.

[16] C. Nentwich, W. Emmerich, A. Finkelstein. Consistency Management with Repair Actions. *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, Portland, USA, May 2003.

[17] L.M. Ni, Y. Liu. Y.C. Lau, A.P. Patil. LANDMARC: Indoor Location Sensing Using Active RFID. *Proceedings of the 1st IEEE International Conference on Pervasive Computing and Communications (PerCom 2003)*, Dallas, USA, March 2003.

[18] Raghu Ramakrishnan, Johannes Gehrke. *Database Management Systems (Third Edition)*, McGraw-Hill Higher Education.

[19] A. Ranganathan, R. H. Campbell, A. Ravi, A. Mahajan. ConChat: A Context-Aware Chat Program. *IEEE Pervasive Computing (vol. 1, no. 3)*, pp. 51-57, Jul-Sep 2002.

[20] A. Ranganathan, J. Al-Muhtadi, RH. Campbell. Reasoning about Uncertain Contexts in Pervasive Computing Environments. *IEEE Pervasive Computing (vol. 3, no. 2)*, pp. 62-70, Apr-Jun 2004.

[21] M. Roman, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, K. Nahrstedt. A Middleware Infrastructure for Active Spaces. *IEEE Pervasive Computing (vol. 1, no. 4)*, pp. 74-83, Oct-Dec 2002.

[22] Bill N. Schilit, Marvin M. Theimer, Brent B. Welch. Customizing Mobile Applications. *Proceedings of USENIX Mobile & Location-Independent Computing Symposium*, Cambridge, USA, Aug 1993.

[23] Albrecht Schmidt, Kofi Asante Aidoo, Antti Takaluoma, Urpo Tuomela, Kristof Van Laerhoven, Walter Van de Velde. Advanced Interaction in Context. *Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing (HUC 1999)*, Karlsruhe, Germany, Sep 1999.

[24] Bryan Scotney, Sally McClean. Database Aggregation of Imprecise and Uncertain Evidence. *Information Sciences—Informatics and Computer Science: An International Journal (Vol. 155, Iss. 3-4)*, pp. 245-263, Oct 2003.

[25] J. P. Sousa, D. Garlan. Aura: An Architectural Framework for User Mobility in Ubiquitous Computing Environments. *Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture*, Montreal, Canada, Aug 2002.

[26] Chang Xu, S.C. Cheung, Cindy Lo, K.C. Leung, Jun Wei. Cabot: On the Ontology for the Middleware Support of Context-Aware Pervasive Applications. *Proceedings of the IFIP Workshop on Building Intelligent Sensor Networks (BISON 2004)*, Wuhan, China, Oct 2004.

[27] Chang Xu, S.C. Cheung, Xiangye Xiao. Semantic Interpretation and Matching of Web Services. *Proceedings of the 23rd International Conference on Conceptual Modeling (ER 2004)*, Shanghai, China, Nov 2004.