Inconsistency Detection and Resolution for Context-Aware Middleware Support[1]

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

Shing-Chi Cheung
Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong, China
scc@cse.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 simulated experiments, respectively.

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

General Terms
Algorithms, Management, Performance, Design

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

1. INTRODUCTION
Pervasive computing environments encompass a spectrum of computation and communication devices, which 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 evolving environments and provide smart services. For example, a mobile phone would vibrate rather than beep in a concert if its application knows its 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 thus poses an impending requirement on *context consistency* management.

In pervasive computing, the *context* of a computation task refers to the circumstance or situation in which the task takes place (e.g., its user's current location and activity). *Context consistency* is maintained when there is no contradiction in the 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 where Peter is working, the room in which Peter is located, and the condition of the 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 an 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-acquired information that Room 3504 is an operating theatre. However, from 3) the agent might draw another conclusion that Peter is not attending an emergency and should be able to help patient Michael immediately if Michael becomes unconscious at that moment. The two opposite assessments reflect the contradiction in the current context, i.e., conflicting understanding to the surrounding environment. Thus the agent might have difficulty in deciding whether to guide Peter for 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 inconsistency by itself due to the lack of adequate reasoning capabilities, global situation assessment and effective repairing actions, which typically require considerable computing resources not available for portable pervasive computing agents.

An unfortunate observation is that context inconsistency is commonly found in real-world applications. *ActiveCampus* is a real-life example [8]. When context inconsistency occurs due to stale data, *ActiveCampus* is unable to correctly estimate a person's location, and this 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 environment can make context easily obsolete** [11]**:** For example, the location context of a fast-moving subject (e.g., a doctor running for an emergency) is prone to error.
- **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**

[1] Crafted version: Made in Feb 2017 (originally published at ESEC/FSE in Sep 2005). Contact: Chang Xu (changxu@nju.edu.cn).

due to computing-resource limitations: 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 failure can lead to incomplete context** [11]**:** The mobility of pervasive computing increases the chance of context loss (e.g., "Peter and Michael enter an operating theatre" vs. "only Peter enters an operating theatre").

The natural existence of such imperfect sources makes context inconsistency a common phenomenon. It is difficult to guarantee correctness, integrity and non-redundancy of context in pervasive computing. However, this problem has not been explicitly addressed in existing context-aware infrastructures (e.g., *Context Toolkit* [6], *EgoSpace* [13], *Gaia* [21] and *Aura* [25]). To address this problem, we identify 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 or 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, and this calls for an automated inconsistency resolution mechanism. Moreover, simple repairing on current context can be inadequate for maintaining a stable running environment for applications. Proactive control on, and feedback to, context sources is necessary for preventing future inconsistencies.

To our best knowledge, in 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] studied in the artificial intelligence (*AI*) field, the similarity lies in that both relate to information inconsistency. The causes of inconsistency and corresponding challenges in resolving it actually differ (see Section 2). Moreover, this paper aims to propose a consistency management framework using software engineering methodology, rather than working on sophisticated inconsistency detection algorithms using *AI* techniques.

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

2. RELATED WORK

Existing work on context-awareness is mostly concerned with either frameworks that support context abstraction or data structures that support context queries. The pioneering work by Schilit et al. [22] proposed using environmental servers to manage context. The context model proposed in the work is simple. Schmidt et al. [23] presented a layered processing model, in which sensor outputs are transformed into cues comprising a set of values with certainty measurements. Gray et al. [7] were concerned with capturing context meta-information, which describes features such as representation, quality, source, transformation and actuation. Harter et al. [9] proposed a conceptual context model, which is constructed using an entity-relationship based language. Henricksen et al. [11] comprehensively analyzed context by covering temporal characteristics, information imperfection, various representations and high interrelation. These pieces of work are 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 infrastructures, e.g., *Gaia* [21], *Aura* [25] and *EasyLiving* [3], have been proposed to provide middleware support for pervasive computing. They are mainly concerned with organization of, and collaboration among, pervasive computing devices and services. Other infrastructures mostly focus on context processing, reasoning and programming support. An earlier piece of representative work is the *Context Toolkit* framework [6]. It assists developers by providing abstract components (e.g., context widgets, interpreters and aggregators), which can be connected together to capture and process context data from sensors. *Context Toolkit* falls short in supporting highly-integrated context-aware applications. To overcome this, Griswold et al. [8] proposed to apply a hybrid mediator-observer pattern in an application's architecture. Henricksen et al. [10] presented a multi-layer framework to support both branching and triggering programming models. Ranganathan et al. [19] discussed how to resolve potential semantic contradictions in context by reasoning based on first-order predicate calculus and Boolean algebra. Later they extended the work to reason about context uncertainty using fuzzy logic [20]. These pieces of work 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 repairing of inconsistent context.

Pervasive computing, a relatively new but fast growing field, shares many observations and technologies with *AI*, active database and software engineering fields. In the *AI* field, expert systems have been developed to support intelligent strategy making. Much effort has been made on the evidence aggregation problem, such that the systems are able to take reasonable strategies on top of contradicting evidence or rules, but the causes of inconsistency are rarely addressed. Composite event detection is an important issue in the active database field for triggering pre-defined actions once desired events are detected. *E-brokerage* [14] and *Amit* [1] are two well-known research projects aiming at detecting composite event occurrences or situation changes with complex timing constraints. The difference lies in that the former is based on event instance modeling and the latter is on event type modeling. In the software engineering field, *CARISMA* [4] was proposed as reflective middleware support for mobile applications. It focuses on policy conflict resolution, which is similar to our work. However, it assumes that accurate context information can be obtained by probing sensors periodically, and this is different from the basis on which our work is built. Nentwich et al. proposed a framework for repairing inconsistent *XML* documents based on the *xlinkit* technology [15]. It generates interactive repairing options from first order logical formulae, which 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 these pieces of work provide similar experience in problem analysis, their

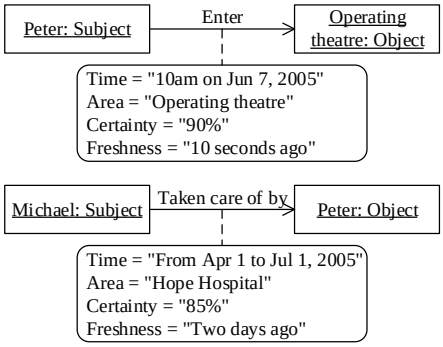

Figure 1. Two context instance examples.

techniques are still inadequate for managing context consistency on two aspects:

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

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 generic data structure is required for context representation. However, we do not adopt a too simple representation like name-value pair or tuple space [13] for the sake of manageability, since it often requires many pairs or tuples to represent a single context piece. On the other hand, neither do we intend to list all context characteristics as proposed by Henricksen et al. [11] due to its high management and computation costs.

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

- *Subject*, *predicate* and *object* specify the content of a context, where *subject* and *object* are related by *predicate* (using a simple English sentence structure), e.g., "Peter (*subject*) enters (*predicate*) an operating theatre (*object*)".
- *Time* and *area* specify temporal and spatial constraints relevant to the context: *time* represents the time point or period in which the context is effective (e.g., "10am on Jun 7, 2005" or "from Apr 1 to Jul 1, 2005"); *area* represents the place to which the context relates (e.g., "Hope Hospital").
- *Certainty* specifies a percentage value evaluating the probability level of the context (e.g., "90%"), and *freshness* specifies 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 represents a context's effective time point or period, while the latter represents a context's generation time. Normally, they are different. For example, context "Michael is taken care of by Peter" may have a long effective period (say, two months), but its generation time can be "two days ago". In pervasive computing, *freshness* is a basic requirement for evaluating context validness, because computing environments tend to change fast and current context can expire quickly. Such consideration is not supported in existing event detection work such like [1] and [14].

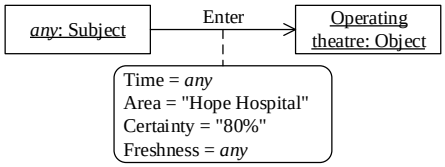

Figure 2. A context pattern example.

For the purpose of context recognition, we introduce two concepts, *context instance* and *context pattern*. 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 value meaning "do not care". Intuitively, each pattern represents a family of context instances.

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

4. MANAGE CONTEXT CONSISTENCY

A key requirement in context consistency management is the ability of bridging the gap between the context recognized by supporting middleware and the inconsistency to which the middleware needs to react. This paper aims to bridge 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.
- **Automated resolution:** Context evolution is dynamic and fast, and this calls for an automated resolution mechanism for detected inconsistency.
- **Feedback control:** Repairing on current context is inadequate, and it is necessary to provide 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 an 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 temporal constraints (e.g., "an operation will be performed *in ten minutes*"), spatial constraints (e.g., "the doctor and patient are *in the same room*") and data constraints (e.g., "*the person* entering the room *is a doctor*").

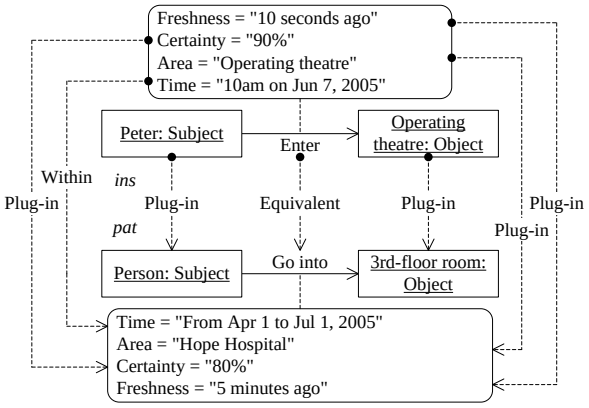

Figure 3. A context matching example.

To model such complex context, we begin with basic blocks (i.e., context instances and patterns in Section 3) and use operations (e.g., *context matching*) to connect them together.

4.1.1 Semantic Context Matching

We introduce a fundamental operation, *context matching*, below. *Context matching* is a process of checking whether a context instance and a pattern match with each other or not. Unlike existing work, 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). Formally,

pat_mat(*ins*, *rules*) :=
  { *pat*∈*Patterns* | ∀*field*.match(*rules.field*, *ins.field*, *pat.field*) };
ins_mat(*pat*, *rules*) :=
  { *ins*∈*Instances* | ∀*field*.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 a certain *unification rule*. The notation of *unification rule* is based on concept semantic relationship [27]. 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:

- *Equivalent*: if $E(c_1) = E(c_2)$;
- *Subsumed*: if $E(c_1) \subset E(c_2)$;
- *Including*: if $E(c_1) \supset E(c_2)$;
- *Disjointed*: 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 the following 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-disjointed relationship), and (6) *unrelated condition* (v_1 and v_2 are disjointed), then v_1 is unifiable with v_2. Otherwise, v_1 is not unifiable with v_2.

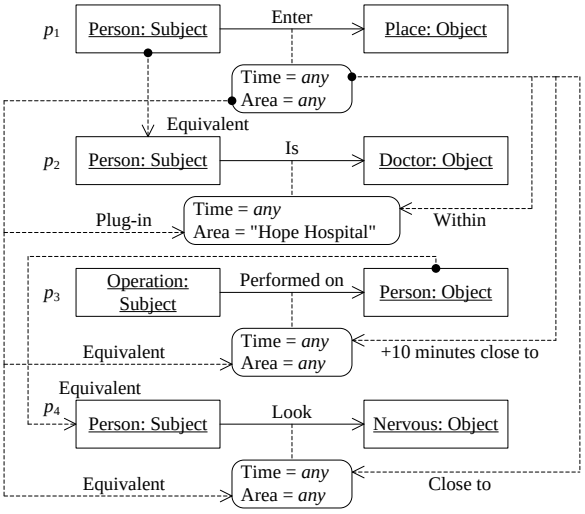

Figure 4. A complex context example.

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

Different fields in a pattern can apply different conditions. Figure 3 illustrates an example, which shows that context instance *ins*, "Peter enters an 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, the *certainty* value "90%" in *ins* is unifiable with the *certainty* value "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 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, we name it *semantic context matching*. In context inconsistency detection (see Section 4.2), automated reasoning can be supported by semantic context matching.

4.1.2 Complex Context and Constraints

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*, pat_1, $field_1$, pat_2, $field_2$), meaning that if there are two context instances matched for pat_1 and pat_2, respectively, their values in corresponding fields $field_1$ and $field_2$, respectively, should satisfy the given *rule* (unification rule). To make the whole complex context *ccx* assessed to be the current situation, there should be a group of context instances matching each pattern in *ccx*, respectively, and these context instances should also satisfy all *ccx*'s constraints.

Figure 4 illustrates the complex context example discussed at the

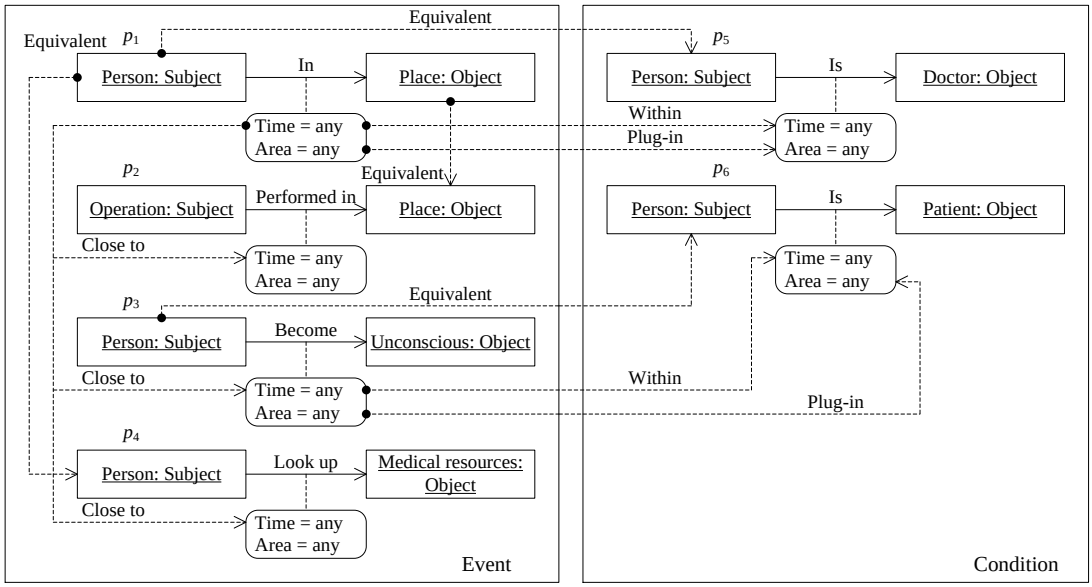

Figure 5. An inconsistency trigger.

beginning of Section 4.1. It consists of four patterns and eight constraints between them (including three temporal constraints, three spatial constraints and two data constraints), represented by dashed lines. We explain three of them for illustration:

- **Temporal constraint ("+10 minutes close to", p_1, "Time", p_3, "Time"):** *The time* when a person enters such a place *is about 10 minutes before* an operation is performed there.
- **Spatial constraint ("Equivalent", p_1, "Area", p_3, "Area")**: A person enters *a place where* an operation is going to be performed in ten minutes.
- **Data constraint ("Equivalent", p_1, "Subject", p_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 joined "columns" are specified by constraints. The difference is that "columns" are related by semantics, in particular when one uses the equivalent condition, which connects two field values of a similar meaning (e.g., "enter" and "go into"). This kind of join is called *semantic-join*.

The semantic matching and join used in our context model is a major difference from existing work. One advantage is that it simplifies the task of specifying generic context inconsistency (e.g., context "*a person* is conducting two *unrelated tasks* 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 our previous model preparation, we in the following introduce 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, which activates the trigger. It specifies complex context *ccx* describing our interested situation. The change occurs when *ccx* is assessed to be the current situation (all patterns match).
- *Condition* is a context-related query, which is run when the trigger is activated. It includes a group of patterns representing a series of tests. Each pattern should match at least one context instance in the context repository (storing history context instances) such that the whole condition is satisfied.
- *Action* is a routine, which is executed when the trigger is activated and its condition is satisfied.

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

4.2.1 Inconsistency Detection Algorithm

There are three context types according to their nature: *sensed contexts* (e.g., "Peter enters an operating theatre") are collected by sensor devices; *domain contexts* (e.g., "Peter takes care of Michael" or "Michelle was born in Jan 1977") are supplied by human operators; *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 new timestamped context events (mainly sensed or derived contexts). Step (2) conducts queries upon stored history context instances (mainly domain contexts).

Context nature is a factor affecting the execution of inconsistency triggers. The size of the detection buffer (or matching queues) is decided by freshness needs of concerned patterns in Step (1). To save memory, usually only sensed and derived contexts are monitored in this step. Such contexts tend to have a strong freshness need, leading to short matching queues.

Compared to traditional event detection, inconsistency detection

needs to consider various types of constraints (e.g., temporal, spatial and data constraints, as discussed earlier). Even for temporal constraints, inconsistency detection has to differentiate a context's generation time from its effective time, while traditional event detection focuses only on an event's occurrence time. From the perspective of temporal constraints, an event's occurrence time is analogous to a context's generation time. So context inconsistency detection can subsume event detection. We give our detection algorithm below:

```
(1) Context preprocessing thread (t1):
    wait for new 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 timer t for pat based on pat's
          freshness need and ins' generation time
(2) Inconsistency triggering thread (t2):
    wait for 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) Time controller thread (t3):
    wait for expired timer t
    remove the first element from t's related
      pattern pat's matching queue pat_que
    if pat_que is empty
      then remove t
      else update t based on pat's freshness need and
        the new first element's generation time
```

The algorithm consists of three parts. Thread t_1 conducts matching for each new context instance, and attaches a copy of it to each matched pattern's matching queue. Thread t_2 monitors all matching queues to see whether there is a group of context instances able to activate a trigger with all its constraints and conditions satisfied. Threads t_1 and t_2 work as a producer-consumer pair of context instances for inconsistency triggering purposes. Thread 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, our algorithm adopts the *continuous* policy [1], i.e., maximizing the use of each context instance within its freshness need's scope. The freshness need of a pattern specifies the period in which a matched context instance for this pattern stays valid. Under this configuration, the algorithm detects all possible inconsistencies among valid context instances. In implementation, freshness needs can be enforced by timers.

The continuous policy for event detection is generally impractical due to its unlimited memory cost, but such policy is feasible for our complex context detection. This is because one can restrict the memory cost by setting a reasonably strong freshness need, i.e., a short time period. The maximal memory cost of our implementation is below 23MB (including Sun JRE'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 concerned patterns' freshness needs. Fortunately, the delay time is also controllable (fully decided at design time by specifying freshness need). Users are suggested to avoid using unreasonably weak freshness needs in Step (1). Weak freshness needs should be moved to Step (2), which does not affect the delay time. Another solution is to allow applications to access temporary context data (still in matching queues) at the cost of possible inconsistency.

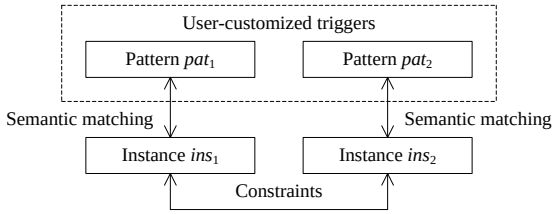

Figure 6. Context matching and inconsistency triggering.

4.2.2 Inconsistency Resolution

The context matching and inconsistency triggering model contributes to inconsistency detection by semantically specifying and detecting: (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 to be taken to ensure the accuracy of concerned contexts.

Generally, when one detects inconsistency between new and old data stored in an information repository, common actions are to repair the repository based on two 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 focuses solely on repairing the repository, but pays little attention to repairing concerned context sources. As such, the environment may still keep generating inconsistent contexts.

Recently, a substantial amount of work has been proposed on active systems, which either react automatically to environmental changes (i.e., reactive systems) or predict changes in their environments (i.e., proactive systems) [1]. Concerning inconsistency resolution, traditional accept/reject policies belong to reactive repairing actions, which work when 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 conducted to repair context data in the context repository. This is analogous to accept/reject policies except that we also support on-demand context update.
- *Proactive repairing actions* are conducted to repair context sources, e.g., to control or adjust problematic sensing devices to avoid further occurrences of inconsistent contexts.

Two policies are supported in reactive repairing actions:

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

2. Dynamic policy:
- Delete the most uncertain context instances (e.g., delete the instance with the lowest uncertainty).
  Primitive: `delByUct(LOWEST, {pid1, …, pidn})`.
- Re-query concerned context sources to obtain a new copy for certain context instances.

Primitive: `uptByPat(pid, queryTime).`

By default, all context instances kept in the detection buffer will be moved to the context repository automatically when concerned timers expire except for those that have to be deleted according to the chosen policy. For the dynamic policy, the query time has to be enforced when executing `uptByPat`. Such time enforcement is usually reasonable and useful as discussed in related studies [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 complete, and that the user desires the location provider to expend a sufficient amount of effort to locate a certain person. Although a too long time limit is unacceptable for timely resolution of inconsistency, multi-threading for parallel processing of inconsistencies can alleviate this problem.

Two policies are also supported in proactive repairing actions:

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

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

Most sensor devices and software programs support direct control from middleware on their lifecycles. This 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(p1,500)
(2) uptByPat(p4,500)
(3) int pid = delByUct(LOWEST,{p1,p4})
```

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

Supporting proactive repairing actions can be non-trivial. Different context sources may vary in the support of inconsistency resolution, and the user 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.

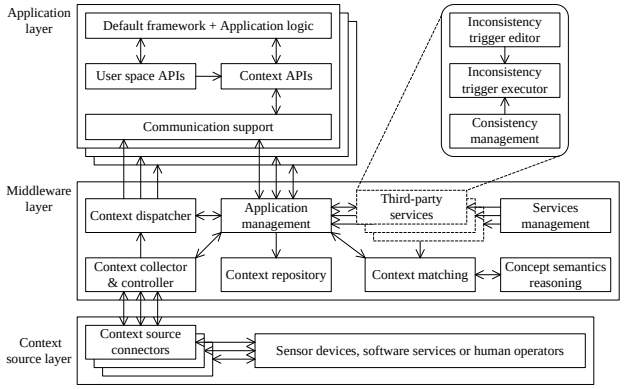

Figure 7. The *Cabot* system architecture.

5. IMPLEMENTATION

Our consistency management framework assumes the availability of an underlying context middleware. We implemented the framework based on one of our research projects – *Cabot* [26]. *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*, *middleware layer* and *context source layer* (Figure 7).

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 in *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 component in the framework enables developers to customize their inconsistency triggers. Repairing actions are also specified at design time. Currently, they are implemented through a callback mechanism in terms of user-designed Java classes, which use our framework's built-in primitives (see Section 4.2.2). The framework is responsible for maintaining a consistent context repository. Applications access contexts 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 [15], 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 recognition and inconsistency detection. The new detection algorithm is based on the *Amit* technology [1] (see Section 8).

6. CASE STUDY

We take an automated vehicle (*AV*) system based on the Radio Frequency Identification (*RFID*) technology as a case study. The *AV* system is one of our ongoing projects on context-awareness with a goal of providing continuous remote control on intelligent vehicles driving in an adverse environment (e.g., too dark, dangerous, hot or noisy).

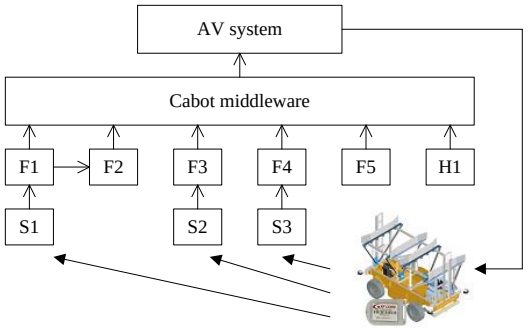

Figure 8. The *AV* system and context source deployment.

To facilitate location estimation for vehicles, some reference sites were chosen and installed with *RFID* tags. These tags together with those attached to vehicles were used for tracing each vehicle, routing them to conduct designated tasks at different destinations. The *AV* system is context-aware in that it controls vehicles based on environmental contexts and each vehicle's conditions. Its typical tasks include automated path selection and collision avoidance.

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

As a result, context inconsistency naturally occurs in reality and affects correct functioning of the *AV* system. For example, automated 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 calculated. 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 calculate context data. The *AV* system's strategies may be unexpectedly affected by inconsistent contexts and possibly generate incorrect control on vehicles.

Suppose that the following context sources have been set up (*S*: sensor device, *F*: software program, *H*: human operator (Figure 8):

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

We consider two major functions of the *AV* system:

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

The *SELT* function may be affected by vehicle location calculation, which is not always accurate (e.g., *LANDMARC* has an average error of 1 meter under the experimental setting discussed in [17]). The *CTRL* function depends much on 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 at different areas"). To alleviate the impact of context inconsistency, the following two inconsistency triggers are designed:

***SELT*:** If a vehicle's continually calculated 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 include: updating the latest location (enforcing query time less than 1 second) and deleting the old one if they differ largely.
***CTRL*:** If two collision reports from *RFID*-based and ultrasonic-based technologies are inconsistent, update the latter. If they are still different, choose the former and increase the inconsistency counter for the latter. If the counter value reaches 5, restart the concerned ultrasonic sensor. If the value has been already larger than 10, stop the sensor and write a system log for suggested maintenance (possibly damaged).

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

7. PERFORMANCE MEASUREMENT

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

Standby world: This is an empty scenario, which 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 incoming contexts activates the 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 incoming contexts activates the patterns in inconsistency triggers. However, the conditions of a high percentage (67%) of 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 incoming contexts activates the patterns in inconsistency triggers, and the conditions of a high percentage (67%) of activated inconsistency triggers are satisfied. The inconsistency

Table 1. Performance measurement results.

	Sta. world	Noi. world	Fil. world	Com. world
Incoming ctx.	2,000	2,000	2,000	2,000
Activated tgr.	0	39	974	1,310
Triggered inc.	0	69	1,632	2,250
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
Ctx. (/m)	62,565	5,008.35	201.82	148.26
Inc. (/m)	0	172.79	164.69	166.79

triggers are very complex (i.e., conditions are tested with constraints).

Our experiments were designed for comparing *Cabot*'s performance in the four simulated worlds. They were conducted on a Pentium IV @3.2GHz machine running Microsoft Windows XP Professional. A context source thread sent 2,000 context instances to *Cabot* at a pace of 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 to 6 patterns with 11 constraints (if any). For contrast, the repository contained a fixed number (100) of history context instances for condition testing for activated inconsistency triggers. All freshness needs of patterns in inconsistency triggers were set to 10 seconds. Three parameters were monitored:

(1) The numbers 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) The total time, event pattern matching time, condition pattern matching time, constraint enforcement time and other overhead time (all in second).
(3) The numbers of processed context instances and of detected inconsistency number (both per minute).

Table 1 presents averaged results of performance measurement for five executions with little difference among them, which show:

(1) *Cabot*'s upper bound was about 62,500 context instances per minute. This rate was achieved when none of incoming context instances takes part in any inconsistency detection.
(2) *Cabot*'s 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 context instances) is computationally expensive. The reason is that our current implementation cannot utilize mature database technologies, which do not support semantic matching and join.

8. DISCUSSIONS

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

The approach adopted by *Amit* is closer to ours. It is based on event type modeling since any event instance belonging to a relevant event type can participate in its targeted situation detection. In order to cope with complex context detection in pervasive computing, *Amit*'s underlying data structures have to be modified to allow for more attributes such as *effective time* and *area* such that complex temporal, spatial and data constraints can be modeled. Moreover, the detection algorithm has to be modified to enforce new complex constraints such as freshness need. 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 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 has outperformed its previous version by 3700%, 450%, 130% and 150% under the four simulated worlds, respectively.

9. CONCLUSION AND FUTURE WORK

In this paper, we have studied the natural imperfectness of context in pervasive computing environments, and analyzed the challenges of context consistency management from two aspects: inconsistency detection and resolution. A formal semantic matching and inconsistency triggering model is proposed to recognize inconsistent contexts. Then a proactive repairing mechanism is proposed to realize automated inconsistency resolution. 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 inconsistencies is impractical. So we are also working on incremental violation checking techniques for consistency constraints, which 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 from the Research Grants Council

of Hong Kong (Project No. HKUST6167/04E). The authors would like to thank Michael Liu for the case study from his led AV team.

REFERENCES

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

[2] P. Bahl, V. N. Padmanabhan and 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 and S. Shafer. EasyLiving: Technologies for Intelligent Environments. In *Proceeding of the 2nd International Symposium on Handheld and Ubiquitous Computing*, Bristol, England, 2000.

[4] L. Capra, W. Emmerich and C. 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 and D. Mishra. Snoop: An Expressive Event Specification Language for Active Databases. *Data Knowl Eng 14.1*: pp. 1-26, 1994.

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

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

[8] W.G. Griswold, R. Boyer, S.W. Brown and T.M. Truong. A Component Architecture for an Extensible, Highly Integrated Context-Aware Computing Infrastructure. In *Proceedings of the 25th International Conference on Software Engineering*, Portland, USA, May 2003.

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

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

[11] K. Henricksen, J. Indulska and A. Rakotonirainy. Modeling Context Information in Pervasive Computing Systems. In *Proceedings of the 1st International Conference on Pervasive Computing*, Zurich, Switzerland, Aug 2002.

[12] G. Judd and P. Steenkiste. Providing Contextual Information to Pervasive Computing Applications. In *Proceedings of the 1st IEEE International Conference on Pervasive Computing and Communications*, Dallas, USA, Mar, 2003.

[13] C. Julien and G.C. Roman. Egocentric Context-aware Programming in Ad Hoc Mobile Environments. In *Proceedings of the 10th International Symposium on the Foundations of Software Engineering*, Charleston, USA, Nov 2002.

[14] A.K. Mok, P. Konana, G. Liu, C.G. Lee and H. 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 and 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 and A. Finkelstein. Consistency Management with Repair Actions. In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, Portland, USA, May 2003.

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

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

[19] A. Ranganathan, R. H. Campbell, A. Ravi and A. Mahajan. ConChat: A Context-Aware Chat Program. *IEEE Pervasive Computing 1(3)*, pp. 51-57, Jul-Sep 2002.

[20] A. Ranganathan, J. Al-Muhtadi and R.H. Campbell. Reasoning about Uncertain Contexts in Pervasive Computing Environments. *IEEE Pervasive Computing 3(2)*, pp. 62-70, Apr-Jun 2004.

[21] M. Roman, C. Hess, R. Cerqueira, A. Ranganathan, R.H. Campbell and K. Nahrstedt. A Middleware Infrastructure for Active Spaces. *IEEE Pervasive Computing 1(4)*, pp. 74-83, Oct-Dec 2002.

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

[23] A. Schmidt, K.A. Aidoo, A. Takaluoma, U. Tuomela, K.V. Laerhoven and W.V. de Velde. Advanced Interaction in Context. In *Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing*, Karlsruhe, Germany, Sep 1999.

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

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

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

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