Heuristics-Based Strategies for Resolving Context Inconsistencies in Pervasive Computing Applications[1]

Chang Xu* Shing-Chi Cheung* Wing-Kwong Chan# Chunyang Ye*
* Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong, China
# Department of Computer Science, City University of Hong Kong, Kowloon Tong, Kowloon, Hong Kong, China
{changxu, scc}@cse.ust.hk, wkchan@cs.cityu.edu.hk, cyye@cse.ust.hk

ABSTRACT
Context-awareness allows pervasive computing applications to adapt to changeable computing environments. However, contexts, the pieces of information that capture the characteristics of environments, are often error-prone and inconsistent due to noise. Various strategies have been proposed to enable automated context inconsistency resolution. However, they are formulated on different assumptions that may not hold in practice. This causes applications to be less context-aware to different extents. In this paper, we investigate such impact and propose our new resolution strategy. We conducted experiments to compare our work with major existing strategies. The results showed that our strategy is both effective in resolving context inconsistencies and promising in its support of applications using contexts.

Keywords
Context-awareness, context inconsistency.

1. INTRODUCTION
Various technologies, such as wireless connection, sensor network, and Radio Frequency Identification (RFID), have enabled pervasive computing applications to interact with each other *contextually* (e.g., "reachable via wireless connection", "motion-detectable by a sensor", and "identifiable by RFID signal"). *Contexts* capture the characteristics of computing environments, and applications adapt themselves based on contexts to changeable environments. For example, a smart phone would vibrate rather than beep in a concert hall to avoid disturbing an ongoing performance, but would roar loudly in a foot-ball match to draw its user's attention. Situations, such as "in a concert hall" or "during a football match", are referred to as *contexts*, and the smart phone's capability of adapting to different environments is referred to as *context-awareness.*

Contexts are often noisy [8][14]. Noisy contexts can cause *context inconsistencies* [17], which imply conflictions among contexts, when contexts violate predefined constraints derived from application requirements. On the other hand, contexts change frequently [6] (e.g., new location contexts are continuously produced as people move around, and existing temperature readings of a room become obsolete as time passes). Human participation in manually resolving context inconsistencies is inefficient and thus infeasible. To resolve context inconsistencies automatically, various strategies have been proposed in the literature. Bu et al.[1] proposed discarding all contexts relevant to any inconsistency. Chomicki et al. [4] suggested discarding the latest inputted event (context) if it causes any conflicting actions (inconsistency), or randomly discarding some actions. Ranganathan et al. [13] and Insuk et al. [7] attempted to follow user preferences or policies to resolve inconsistencies in an application's situation evaluation.

Regarding the preceding resolution strategies, we study the following research questions: *Do existing resolution strategies satisfactorily resolve inconsistent contexts for pervasive computing applications? Under what assumptions in pervasive computing environments, can one find a better (reliable), automated resolution strategy? Finally, what are the observations and issues in the search for an optimal resolution strategy?*

In this paper, we assume the existence of a middleware infrastructure that collects contexts from distributed context sources (e.g., hardware sensors for location estimation or software programs for user action reasoning), and manages these contexts for pervasive computing. We study inconsistency resolution strategies as a management service in the middleware. The rest of this paper is organized as follows. Section 2 reviews existing resolution strategies and analyzes their limitations. Section 3 presents our new resolution strategy. Section 4 conducts experiments to compare all these strategies and evaluate their impact on context-aware applications. Section 5 discusses learned lessons and related research issues. Finally, Section 6 presents related work in recent years and Section 7 concludes the paper.

2. INCONSISTENCY RESOLUTION STRATEGIES
In this section, we review and compare existing automated strategies for context inconsistency resolution.

2.1 Illustrative Example
Our comparison uses a location tracking application example. In pervasive computing, location context has been widely studied and used in context-aware applications. Inconsistencies in location contexts occur when a person's location changes in a way violating certain *consistency constraints* [11]. Such constraints check whether the person's location falls into a feasible area (depending on whether the person is permitted into this area), or whether the person's location change is too large such that a reasonable velocity limit is breached (e.g., a person "jumps" from one floor to another in a very short period of time). Suppose that in one application, Peter walks steadily at an average velocity of v over one period, and the application requires that *Peter's walking velocity estimated from his location changes should be less than 150% of* v. Here, 150% is selected for error tolerance.

Figure 1 shows a segment of Peter's traversal path with five tracked locations, from d_1 to d_5. They are location contexts calculated chronologically by a location tracking application (e.g., Landmarc algorithm [12]). Suppose that the dashed curve is the actual path

[1] Crafted version: Made in Feb 2017 (originally published at ICDCS in Jun 2008). Contact: Chang Xu (changxu@nju.edu.cn).

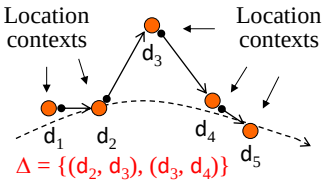

Figure 1. Five example location contexts.

Peter walks, which differs from the estimated path composed of tracked locations. This is common due to inaccuracy of existing location tracking techniques in the literature.

Suppose that the application specifies a consistency constraint on possible changes of location contexts. We first investigate location changes formed by adjacent locations. In Figure 1, location d_3 seriously deviates from the actual path, and therefore we assume that two adjacent location pairs (d_2, d_3) and (d_3, d_4) do not satisfy the constraint (i.e., Peter's walking velocity estimated from the two location pairs is not within the range of 150%·v). As a result, two location context inconsistencies are detected and they are illustrated by set Δ in Figure 1.

2.2 Drop-latest Resolution Strategy

In the drop-latest resolution strategy [4], the latest context leading to an inconsistency would be discarded. This strategy assumes that the collection of existing contexts is consistent, and that any new context is permitted to enter this collection only if the new context does not cause any inconsistency with existing contexts.

Figure 2 shows two example scenarios using this strategy. In Scenario A (the same as in Figure 1), inconsistency (d_2, d_3) is detected and context d_3 is discarded as it is the latest. Since d_3 has been discarded, inconsistency (d_3, d_4) no longer occur. In this case, the strategy correctly discards d_3 for Scenario A. However, in Scenario B, context d_3 is closer to d_2 than it is in Scenario A, such that the location pair (d_2, d_3) no longer violate the consistency constraint. As a result, the first detected inconsistency becomes (d_3, d_4), and context d_4 instead of d_3 is discarded since it is the latest. In this case, the result is an incorrect resolution.

The drop-latest resolution strategy fails in Scenario B because even if context d_3 does not cause inconsistency with context d_1 or d_2, it may still be incorrect. Determining context d_3 immediately to be correct (when no inconsistency among contexts d_1, d_2, d_3 is detected) would cause the subsequent inconsistency (d_3, d_4). Then context d_4 would be mistakenly discarded since this strategy always discards the latest context that causes inconsistency. This result suggests that *determining a context immediately to be correct or incorrect based on existence or non-existence of a single inconsistency may not give a desirable resolution*.

2.3 Drop-all Resolution Strategy

In the drop-all resolution strategy [1], all contexts leading to an inconsistency are simply discarded. The strategy assumes that all contexts relevant to inconsistency are incorrect and thus discards all of them for safety.

Figure 3 shows this strategy's resolution results for the same two scenarios. In Scenario A, inconsistency (d_2, d_3) is detected and then both contexts d_2 and d_3 are discarded. As a result, inconsistency (d_3, d_4) no longer occur since context d_3 has been discarded. Discarding context d_3 is a correct resolution, but context d_2 is lost at the same time. In Scenario B, inconsistency (d_3, d_4) is detected, and then both

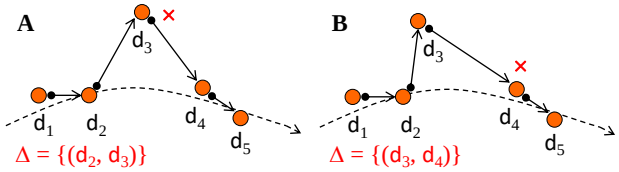

Figure 2. Drop-latest resolution strategy.

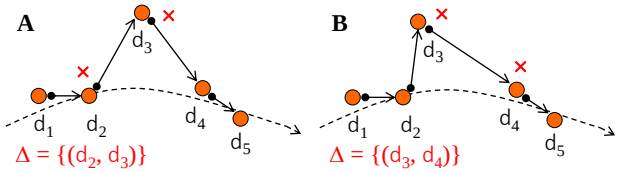

Figure 3. Drop-all resolution strategy.

contexts d_3 and d_4 are discarded. This strategy also causes context loss of d_4, which is actually correct.

We observe that the drop-all resolution strategy does not work satisfactorily, because it accepts correct contexts with its overcautious nature. *This nature makes the strategy tend to discard more contexts than necessary*. If discarded contexts are important to an application, the strategy would seriously impact the application (e.g., some key context-aware actions cannot be taken since relevant contexts have been discarded).

The drop-random and user-specified resolution strategies also have unreliable results (depending on random choices and user policies). We do not give concrete examples here due to page limit.

3. DROP-BAD RESOLUTION STRATEGY

In this section, we present our new drop-bad resolution strategy.

3.1 Example Revisited

Suppose that one does not resolve detected context inconsistencies immediately. Instead, one records the count for every context that participates in inconsistency. Then one can obtain a list of count values, each of which tells how many inconsistencies a particular context has ever participated in. For illustration, we calculate count values for the aforementioned Scenarios A and B in Figure 4. In Scenario A, two inconsistencies (d_2, d_3) and (d_3, d_4) are detected without immediate resolution. We observe that context d_3 has a count value of 2 since d_3 participates in both inconsistencies, and that contexts d_2 and d_4 both have a count value of 1 since they participate in only one inconsistency each. Other contexts' count values are zero since they are not involved in any inconsistency. Count values in Scenario B can also be calculated in the same way.

We observe that in Scenario A, context d_3 carries the largest count value (2) among all contexts (i.e., d_2, d_3, d_4) relevant to the two detected inconsistencies, and that it is actually the incorrect context. We conjecture that if a context carries a relatively large count value, it tends to be incorrect. In Scenario B, this conjecture still holds: context d_3 also carries the largest count value (1), but another context d_4 happens to have the same count value (1). Since there is only one inconsistency (d_3, d_4) under consideration, one cannot dig out more useful information to distinguish contexts d_3 and d_4. Then, what if more inconsistency information is available?

We now refine the preceding consistency constraint to allow it to examine more location pairs, which are not necessarily adjacent.

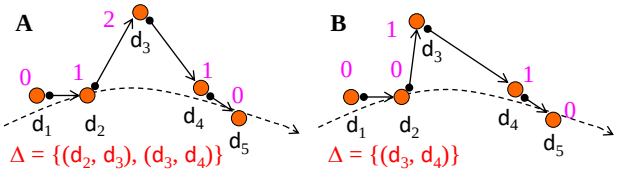

Figure 4. Count values for the two scenarios (1).

That is, it would check whether Peter's walking velocity estimated from location pairs that are separated by one intermediate location, such as (d_1, d_3) and (d_2, d_4), also satisfies the velocity condition. Then, two previous context inconsistencies are still detectable, but now more context inconsistencies could be detected. Suppose that there are two more inconsistencies (d_1, d_3) and (d_3, d_5) in Scenario A and one more inconsistency (d_3, d_5) in Scenario B, as illustrated in Figure 5. With all these detected context inconsistencies, we observe that in both scenarios, context d_3 now carries the largest count value (4 and 2, respectively), and that d_3 is actually the incorrect location context we expect to discard.

This example revisiting makes an interesting observation, which serves as the starting point of our new proposal for automated inconsistency resolution: *A context that participates more frequently in inconsistencies is likelier to be incorrect*. We refer to our new proposal as *the drop-bad resolution strategy* and explain it in detail below.

3.2 Tracked Context Inconsistencies

Unlike existing resolution strategies that immediately resolve any detected context inconsistency, our drop-bad resolution strategy would tolerate the inconsistency until any context participating in this inconsistency is actually used by an application. This strategy keeps track of all context inconsistencies that have been detected but not resolved yet. Let C be the set of contexts. The set of the *tracked context inconsistencies* is represented by $\Sigma \subseteq \wp(\wp(C))$. For example, $\Sigma = \{\{d_3, d_4\}, \{d_3, d_5\}\}$ for the earlier Scenario B in Figure 5 represents that two context inconsistencies are detected but not resolved yet. The two inconsistencies are caused by context pairs (d_3, d_4) and (d_3, d_5), respectively.

Given a set of tracked context inconsistencies, we introduce a count function that returns a list of count values. Every count value indicates how many inconsistencies a certain context has participated in. Let N be the set of natural numbers. Function count is defined as: $\Sigma \rightarrow \wp(C \times N)$. For example, when $\Sigma = \{\{d_3, d_4\}, \{d_3, d_5\}\}$, count($\Sigma$) would return $\{(d_3, 2), (d_4, 1), (d_5, 1)\}$.

The set of tracked context inconsistencies is dynamic due to continual context changes. There are two types of context change. One is *context addition* change, which means that a new context is recognized and checked by the middleware. The other is *context deletion* change, which means that an existing context is now being used by an application and thus needs a decision on whether it is correct or not. We note that context deletion only "removes" a context from the checking of its involved inconsistencies (if any). The context is still available until it expires according to its own available period. We discuss the impact of a context change on the set of tracked context inconsistencies below, as shown in Figure 6:

– When a context addition change occurs, a new context is recognized and checked against consistency constraints. If the context causes any inconsistencies with existing contexts, Σ is added with these newly detected inconsistencies.

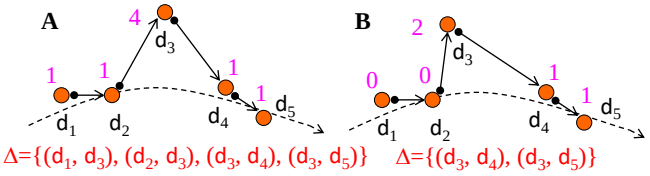

Figure 5. Count values for the two scenarios (2).

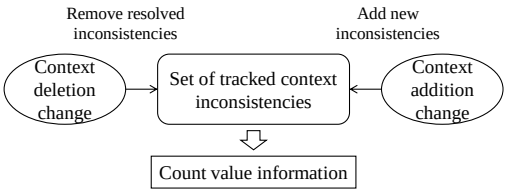

Figure 6. Tracked context inconsistencies.

– When a context deletion change occurs, an existing context is to be used by an application. Any tracked inconsistency involving this context (i.e., those inconsistencies this context has participated in) needs a resolution on whether this context is correct or not. These resolved context inconsistencies are then removed from Σ (i.e., resolved and no need for further tracking).

Whenever the set Σ of tracked context inconsistencies changes, the return value of the count function would be updated accordingly. We explain how to resolve detected context inconsistencies based on the count value information in Section 3.3.

The dynamic maintenance of the set of tracked context inconsistencies allows a context inconsistency to be resolved with additional count value information, as compared to existing resolution strategies. At the same time, it does not prevent applications from using resolved contexts.

For example, in the earlier Scenario A of Figure 5, context d_3 causes two inconsistencies with contexts d_1 and d_2, respectively. Neither context inconsistency is resolved immediately in the drop-bad resolution strategy. The strategy does not decide whether context d_3 is correct or not until subsequent contexts, d_4 and d_5, arrive and are checked. Then two more context inconsistencies (d_3, d_4) and (d_3, d_5) are detected, and the strategy decides with more confidence that context d_3 is incorrect.

3.3 Resolution Process

The resolution process of our drop-bad strategy consists of two parts (see Figure 7). The first part handles the case that a new context is recognized by the middleware (i.e., handling the context addition change). The second part handles the case that a context involved in any tracked context inconsistency is used by an application (i.e., handling the context deletion change).

We explain the resolution process using a context's life cycle, as shown in Figure 8. Each context has one of four states at a time: undecided, consistent, bad, or inconsistent.

When a new context is recognized, its state is initialized to undecided. If the context is irrelevant to any consistency constraint, it is directly set to consistent and made immediately available to applications (Part 1 in Figure 7). This is because no context inconsistency would involve this context. Otherwise, the context is moved to a buffer for further checking. There are two cases (Part 2 of Figure 7):

```
Part 1: When a new context is recognized
recognize a new context d
if d is irrelevant to any consistency constraint
  then d is directly set to consistent and made
    available to applications
  else d is moved to a buffer for tracking any
    inconsistency involving d
```

```
Part 2: When a context is used
wait for context d in the buffer being used
if d is bad or there is inconsistency IC such
  that d is involved in IC and has the largest
  count value among all involving contexts
  then d is set to inconsistent and discarded
  else d is set to consistent and made available
    to applications
for each inconsistency IC′ d participates in
  let d′ be the context also involved in IC′ and
    having the largest count value among all
    involving contexts
  if d′ ≠ d then d′ is set to bad
```

Figure 7. Drop-bad resolution process.

- **Case 1: The context is used by an application after some time.** If the count value of this context is the largest among all contexts participating in any inconsistency, the context is set to inconsistent and then discarded. This is because the context is likeliest incorrect according to our earlier observation. Otherwise, if the "largest count value" condition is not satisfied, then the context is set to consistent, because there must be another context that carries an even larger count value than this context, and this context's involved inconsistency can be resolved later by examining that context.
- **Case 2: The context is affected by other contexts before it is used by an application.** Here, "affected" means that whether this context is inconsistent would be decided earlier than the time when this context is used by an application. Let us consider inconsistency (d_1, d_3) in the earlier Scenario A in Figure 5. Suppose that when context d_1 is used by an application, it carries a count value of 1, which is not the largest (since context d_3 carries a count value of 4). Then, context d_1 would be set to consistent and can be used by the application. Then how to handle inconsistency (d_1, d_3)? Since it has been decided that context d_1 is consistent, then context d_3, which carries the largest count value of 4, should be set to inconsistent to resolve this consistency. We use bad to indicate that a context should be set to inconsistent and discarded later, when it is eventually used by an application.

Regarding this drop-bad resolution process, a question may arise naturally: *Why not discard bad contexts (e.g., context d_3 in the Scenario A) immediately?* We have the following three considerations:

- State bad means that the concerned context would be discarded eventually. Therefore, there is no negative effect by setting the context to bad first.
- Setting a context to bad respects its life cycle because the context has not been used by any application yet.
- Setting a context to bad first and then inconsistent enables the middleware to use the additional time to collect more count value information before discarding this context.

For example, in Scenario A, if one discards context d_3 immediately at the time when context d_1 is used, then the count value information about inconsistencies (d_3, d_4) and (d_3, d_5) would not be available. As a result, the effectiveness of the drop-bad resolution strategy could be compromised. Therefore, the notion of "bad context" tends to collect potentially more count value information for the drop-bad resolution strategy.

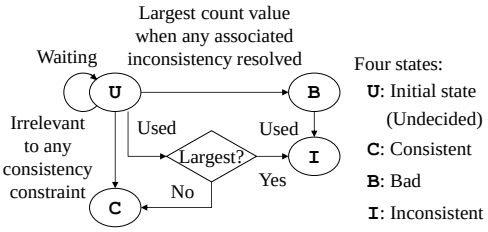

Figure 8. A context's lift cycle.

3.4 Generic Reliability

We note that our drop-bad resolution strategy is applicable to context inconsistencies caused by different types and numbers of contexts (i.e., not only streaming locations or context pairs). In this section, we study the reliability of our strategy for generic context inconsistencies. We refer to a context as *corrupted* if it is incorrect and should be identified as inconsistent. Otherwise, it is *expected*. Note that whether a particular context is corrupted or expected is unknown to any practical resolution strategy in advance.

From our earlier observation, we have the following two heuristic rules:

- ***Heuristic Rule 1.*** *A set of expected contexts does not form any inconsistency.*
- ***Heuristic Rule 2.*** *If a set of contexts forms an inconsistency, then every corrupted context has a larger count value than that of any expected context in this set.*

By Rule 1, we essentially assume that all consistency constraints are correct. In other words, there is no false report of context inconsistency. Rule 2 formulates our earlier observation by assuming that *all corrupted contexts* tend to participate more frequently in context inconsistency than expected contexts. This formulation may be stronger than necessary in some cases. So we formulate a relaxed version as follows:

- ***Heuristic Rule 2'.*** *If a set of contexts forms an inconsistency, then at least one corrupted context has a larger count value than that of any expected context in this set.*

In our drop-bad resolution strategy, a context inconsistency is resolved by discarding some contexts involved in this inconsistency until it vanishes. The key question is whether each thus discarded context is indeed a corrupted one. The following theorem provides a theoretical foundation for this strategy:

Theorem 1. *With the two heuristic rules (1 and 2) holding, the drop-bad resolution strategy is always reliable, i.e., each discarded context is indeed a corrupted context.* □

Since in the strategy, only the contexts carrying the largest count values are discarded, these contexts should be corrupted according to the relaxed Rule 2'. Therefore, we can further formulate the following theorem:

Theorem 2. *With the two heuristic rules (1 and 2') holding, the drop-bad resolution strategy is always reliable, i.e., each discarded context is indeed a corrupted context.* □

We omit the proofs (see our technical report [18]) of Theorems 1 and 2 due to page limit. The two theorems explain how our drop-bad resolution strategy works reliably with generic context inconsistencies (not limited to streaming locations or context pairs). The strategy's reliability helps address problems (e.g., incorrect context

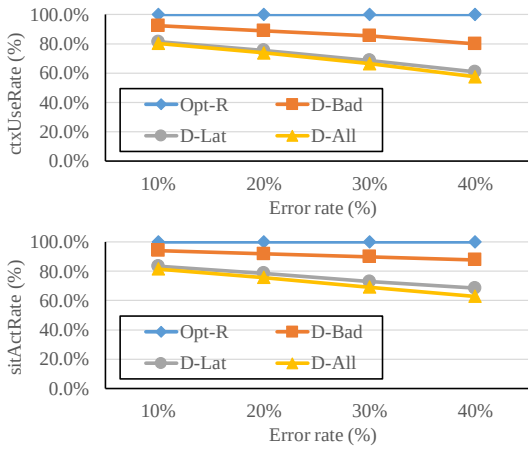

Figure 9. Resolution comparison for the Call Forwarding application.

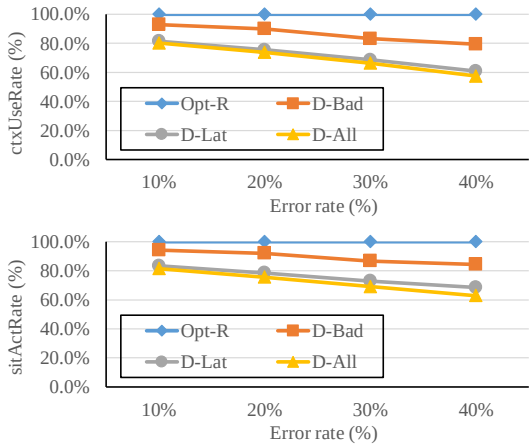

Figure 10. Resolution comparison for the RFID data anomalies application.

discarding and context loss) encountered in existing inconsistency resolution strategies, as we discussed earlier in Section 2.

One natural question may arise: *Is one able to identify all corrupted contexts?* A user normally does not have sufficient and necessary conditions to warrant context consistency for pervasive computing applications. This is analogous to the *test oracle problem*, which makes hidden program faults hard to locate. The full treatment for a better inconsistency resolution strategy needs further study.

4. EXPERIMENTS

We conducted simulated experiments to compare different strategies for context inconsistency resolution in pervasive computing. These strategies are all able to resolve context inconsistency automatically. However, are their resolution results desirable for context-aware applications? Is the context-awareness feature of applications affected during inconsistency resolution? We study these questions in this section.

The experiments extended our preliminary study [20]. In this paper, we report our latest comparison results of three strategies, namely, drop-latest, drop-all, and our drop-bad resolution strategies. These strategies are referred to as D-LAT, D-ALL and D-BAD, respectively. For comparison, we also designed an artificial optimal resolution strategy (OPT-R), which will be explained later in Section 4.1. We conducted a total of 320 groups of experiments, which were evenly distributed to the four resolution strategies with a controlled context error rate for comparison.

The purpose of these experiments is to measure how much a resolution strategy can affect the context-awareness feature of an application (e.g., adaptive behavior based on contexts). Although there is no extensive study on what metrics are most suitable for measuring context-awareness, the use of contexts in applications [1][7] and the activation of target situations [6][13][16] are basic issues in context-aware computing. Therefore, we select the *number of used contexts* and *number of activated situations* as the two metrics for evaluating context-awareness in our experiments. The former measures how many contexts have been actually used by applications, and the latter measures how many situations have been actually activated for applications, after context inconsistency resolution. It can be perceived that any strategy, which discards inconsistent contexts and thus changes the contexts accessible to applications, would certainly affect these two metrics. By comparing values of the two metrics under different strategies, we would be able to infer the extent of impact on the context-awareness feature of applications due to a particular resolution strategy.

4.1 Experimental Setup

We explain our experimental settings below.

Applications, constraints and situations. We selected two applications that were adapted from Call Forwarding [15] and RFID data anomalies [14]. The two applications use location and RFID contexts, which are being widely studied in context-aware applications. We selected five consistency constraints for detecting context inconsistencies and three situations to use contexts for each application. These consistency constraints and situations were from a study we conducted at two universities for collecting user design experience for context-aware applications [19]. To be representative, we selected only those popular constraints and situations. Here, "popular" means that the selected constraints and situations were designed by most participants (university staffs and graduate students) in the study, and therefore they should be easy to think of or widely understandable. The selected constraints and situations cover a large portion of all constraints and situations collected in the study. Here, "Cover" means "identical or semantically equivalent". The coverage is 70.8% and 81.5% for the two applications, respectively.

Contexts and error rates. Contexts were produced by a client thread with a controlled error rate (err_rate) from 10% to 40% with a pace of 10%. These rates were designed based on real-life observations about the RFID error rate from the literature [8][14].

Middleware. The experiments were conducted on the Cabot middleware [16][17] since Cabot supports plug-in context management services. An inconsistency resolution module was implemented as a plug-in service. This module was invoked whenever Cabot received new contexts from the client thread. The module could be enabled with either of the four inconsistency resolution strategies, as mentioned earlier.

Hardware and operating system. We used a machine with an Intel P4 @1.3GHz CPU and 512MB RAM. The operating system is Microsoft Windows XP Professional SP2. The hardware and operating system in our experiments are common for pervasive computing middleware and applications.

Measurements. As mentioned earlier, for comparison purposes,

we assume the existence of an optimal resolution strategy (OPT-R). OPT-R has a specially designed oracle to discard precisely each incorrect context. Therefore, OPT-R serves as a theoretical upper bound of good strategies for context inconsistency resolution. We set the metric values (i.e., number of used contexts and number of activated situations) reported by OPT-R to 100% as the baseline for reference. Then, the metric values reported by other resolution strategies (i.e., D-LAT, D-ALL and D-BAD) were normalized to percentage values against the reference baseline. The two percentages used in our experiments are thus called *context use rate* (ctxUseRate) and *situation activation rate* (sitActRate), respectively. Intuitively, for a context inconsistency resolution strategy, the lower its reported percentage value is, the more an application has been affected by this strategy in context-awareness.

4.2 Experimental Results and Analysis

Figure 9 and Figure 10 compare the four inconsistency resolution strategies for the two applications, respectively (top: context use rate, bottom: situation activation rate). Every point in the figures is associated with a certain strategy (OPT-R, D-BAD, D-LAT or D-ALL) and a certain error rate (10%, 20%, 30% or 40%). The result represented by each point has been averaged over 20 groups of experiments to avoid random error.

We observe that D-LAT and D-ALL are clearly inferior to other inconsistency resolution strategies in that they have reduced the context use rate and situation activation rate by about 20–40%, as compared to the optimal results reported by OPT-R. D-ALL performs the worst since it discarded all contexts involved in any inconsistency and quite a few useful contexts became inaccessible to applications. This greatly lowers its scores in the context use and situation activation rates.

D-BAD performs much better than D-LAT and D-ALL in both metric values, but there is still a gap between the scores achieved by D-BAD and by OPT-R. This shows that focusing on resolving inconsistencies by discarding corrupted contexts only cannot provide adequate support for context-aware applications. We give further discussions and lessons learned later in Section 5.

The experimental results also show that simple inconsistency resolution strategies (e.g., D-LAT and D-ALL) that are based on their built-in heuristics (assumptions) cannot effectively resolve context inconsistencies. As suggested by Figure 9 and Figure 10, the assumptions these strategies rely on do not generally hold in pervasive computing within a wide range of context error rates. D-BAD has used relationships among multiple inconsistencies (in terms of count values) to achieve better scores than the other strategies. This shows that D-BAD has used better, more reasonable heuristic rules.

5. DISCUSSIONS

In this section, we discuss the lessons we learned, further justification to our drop-bad resolution strategy and some related research issues.

5.1 Lessons Learned

The drop-latest and drop-all resolution strategies have built-in selection mechanisms for identifying and discarding certain contexts as being corrupted. The selections are based on their inherent assumptions (heuristics) that are simple and static. The experimental results show that such strategies do not work satisfactorily, and thus suggest that their assumptions may deviate far from what we observed in practical situations. We also find that the existing resolution strategies aim to resolve context inconsistencies without considering possible impact on context-awareness applications. While these resolution strategies can resolve context inconsistencies, the tested applications were seriously affected in the use of contexts and activation of situations.

The drop-bad resolution strategy considers the count value information about multiple inconsistencies that occur within a time window. The strategy favors discarding the contexts that carry the largest count values. This implies that each context the strategy discards tends to relate to more inconsistencies, and thus the strategy essentially aims to resolve context inconsistencies with as few discarded contexts as possible. Nevertheless, this strategy might be trapped at a local optimum. As shown in the experiments, the drop-bad resolution strategy has brought positive results, but still has room for improvement. For example, when the tie case comes, i.e., several contexts carrying the same maximal count value, one needs to carefully examine discarding which particular context among them would cause less impact on context-aware applications. Such examination would potentially bring additional benefits to this strategy. It is part of our ongoing work.

5.2 Further Justification

We observe in the experiments that the drop-bad resolution strategy has used better heuristics in identifying and discarding incorrect contexts than other existing resolution strategies.

Still, we are also conducting a real-life case study to explore how inconsistency resolution strategies would improve the accuracy for location tracking algorithms (e.g., Landmarc [12]). Preliminary results indicate that the drop-bad resolution strategy has high scores for location context survival rate (96.5%, for preventing correct location contexts from being discarded) and removal precision (84.7%, for guaranteeing incorrect location contexts to be discarded) [18]. The study also investigated an interesting question: *How do the heuristic rules discussed in the paper hold in practice?* For Landmarc experiments, our results show that Rule 1 always held, and that Rule 2' (the relaxed version) held in 91.7% cases. We are now on the way to further investigate what percentage value is sufficient for guaranteeing satisfactory results from the drop-bad resolution strategy.

5.3 Related Issues

How does one design correct consistency constraints? Consistency constraints specify necessary properties over contexts for applications. Although sufficient conditions for guaranteeing correctness of all contexts are generally difficult, users are creative in identifying necessary properties of contexts that are relevant to their interests. Our study [19] shows that users were able to identify good consistency constraints, and most of these constraints were similar to each other. Selecting such similar constraints for experiments makes the experiments representative, as we did in Section 4. Besides, in the literature, identifying consistency constraints for software artifacts (e.g., documents, programs, data structures, and class diagrams) has been extensively studied [5][11]. Such experience also applies to contexts (one type of software artifacts). Finally, the correctness of integrating consistency constraints with context-aware applications is being studied by a metamorphic approach [3]. The approach explores the ways of identifying metamorphic relations to build consistency constraints for applications.

How does one decide a time window for the drop-bad resolution strategy? This is important for the strategy to be applicable to generic applications. The time window (i.e., period before a context

is used by applications) can be decided by the context matching time in EgoSpaces middleware [9] and LIME middleware [10], or the checking-sensitive period in the Cabot middleware [16]. Besides, for RFID applications, a time window (either static or dynamic) is already available, usually specified for filtering out unreliable RFID data [8][14]. This time window can also be used for the drop-bad resolution strategy. In the case where new contexts are used immediately by applications, the time window of the drop-bad strategy is trivially reduced to zero. Then the strategy would behave just as the drop-latest strategy. As a result, the effectiveness of the drop-bad resolution strategy would be no worse than those achieved by existing resolution strategies. As future work, the study of impact of time window on the effectiveness of the drop-bad resolution strategy would deserve exploring.

6. RELATED WORK

Consistency management of contexts, an important issue in pervasive computing, is receiving increasing attention. Some research projects focused on application frameworks that support context abstraction or queries [9], middleware infrastructures that work for device organization or service collaboration [13], or programming models that facilitate context-aware computing [10]. New studies have been conducted for the detection and resolution of context inconsistency in pervasive computing.

Researchers have addressed the context inconsistency problem and proposed similar resolution strategies. Bu et al. [1] suggested discarding all conflicting contexts except the latest one, assuming that the latest one has the largest reliability. Insuk et al. [7] and our previous work [16] proposed to resolve context inconsistency by following human choices because they believe that human beings can make the best decision. However, human participation can be slow and unsuitable for dynamic environments.

Some studies are not directly related to context inconsistency but have addressed similar problems on inconsistent or conflicting software artifacts. Capra et al. [2] proposed a sealed-bid auction approach to resolving conflicts among application profiles. In pervasive computing, however, both applications and context generators have no idea whether a particular context is corrupted or not. Chomicki et al. [4] suggested ignoring an inputted event to avoid conflicting actions or randomly discarding some actions to resolve a conflict. Demsky et al. [5] required developers' control in specifying data's default values and the manner to generate new data during data repairing. Nentwich et al. [11] presented a technique to generate optional repair actions to resolve document inconsistency, but users need to select the best choice. Ranganathan et al. [13] suggested setting up rule priorities to follow human preferences. All these approaches attempt to resolve inconsistent or conflicting software artifacts, but they either require human participation or are unsuitable for dynamic pervasive computing environments.

Our approach follows human intuition. Discarding the contexts that participate in most context inconsistencies actually exploits redundancy to identify error. This is similar to some pieces of work that use multiple context-detectors (e.g., multiple localization techniques) to mask error in one technique by redundancy. Our approach is orthogonal to this idea in that it explores and resolves inconsistencies among different types of contexts, in which errors of a single type has been marked. The generic reliability implies that our approach applies to different types and numbers of contexts. This is also different from context filtering techniques that commonly apply to specific applications.

7. CONCLUSION

In this paper, we review several major automated strategies for resolving context inconsistency for pervasive computing, where distributed context sources generate contexts and impact on context-aware applications. We analyze these strategies' limitations and propose our drop-bad resolution strategy. The experiments show that the existing strategies cannot satisfactorily support effective inconsistency resolution, while our strategy has achieved the best results among the reviewed strategies.

Our study is mainly based on a location tracking application because location contexts have been widely used and studied in applications and existing literature. The experiments of Call Forwarding and RFID data anomalies applications show that our observations made in the drop-bad resolution strategy are also applicable to other applications. However, further real-life investigation is needed to study the applicability of our work.

We have also observed a gap between the results offered by the optimal resolution strategy and those reported by the reviewed strategies and ours. This shows room for improvement based on our drop-bad resolution strategy. For example, applying heuristics to context inconsistency resolution should be enhanced with the effort of estimating the impact of a certain resolution strategy on applications and adjusting its resolution action accordingly.

ACKNOWLEDGMENTS

This research was partially supported by the Research Grants Council of Hong Kong under grant numbers 612306, 111107 and HKBU 1/05C, National Science Foundation of China under grant number 60736015, and National Basic Research of China under 973 grant number 2006CB303000.

REFERENCES

[1] Y. Bu, T. Gu, X. Tao, J. Li, S. Chen and J. Lu, "Managing Quality of Context in Pervasive Computing", In *Proceedings of the 6th International Conference on Quality Software*, pp. 193-200, Beijing, China, Oct 2006.

[2] L. Capra, W. Emmerich and C. Mascolo, "CARISMA: Context-Aware Reflective Middleware System for Mobile Applications", *IEEE Transactions on Software Engineering 29(10)*, pp. 929-945, Oct 2003.

[3] W.K. Chan, T.Y. Chen, Heng Lu, T.H. Tse and S.S. Yau, "Integration Testing of Context-Sensitive Middleware-Based Applications: A Metamorphic Approach", *International Journal of Software Engineering and Knowledge Engineering 16(5)*, pp. 677-703, 2006.

[4] J. Chomicki, J. Lobo and S. Naqvi, "Conflict Resolution Using Logic Programming", *IEEE Transactions on Knowledge and Data Engineering 5(1)*, pp. 244-249, Jan/Feb 2003.

[5] B. Demsky and M. Rinard, "Data Structure Repair Using Goal-Directed Reasoning", In *Proceedings of the 27th International Conference on Software Engineering*, pp. 176-185, Louis, USA, May 2005.

[6] K. Henricksen and J. Indulska, "A Software Engineering Framework for Context-Aware Pervasive Computing", In *Proceedings of the 2nd IEEE Conference on Pervasive Computting and Communications*, pp. 77-86, Orlando, USA, Mar 2004.

[7] P. Insuk, D. Lee and S.J. Hyun, "A Dynamic Context-Conflict Management Scheme for Group-Aware Ubiquitous

Computing Environments", In *Proceedings of the 29th Annual International Computer Software and Applications Conference*, pp. 359-364, Edinburgh, UK, Jul 2005.

[8] S.R. Jeffery, M. Garofalakis and M.J. Frankin, "Adaptive Cleaning for RFID Data Streams", In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pp. 163-174, Seoul, Korea, Sep 2006.

[9] C. Julien and G.C. Roman, "EgoSpaces: Facilitating Rapid Development of Context-Aware Mobile Applications", *IEEE Transactions on Software Engineering 32(5)*, pp. 281-298, May 2006.

[10] A.L. Murphy, G.P. Picco and G.C. Roman, "LIME: A Coordination Model and Middleware Supporting Mobility of-Hosts and Agents", *ACM Transactions on Software Engineering and Methodology 15(3)*, pp. 279-328, Jul 2006.

[11] C. Nentwich, W. Emmerich and A. Finkelstein, "Consistency Management with Repair Actions", In *Proceedings of the 25th International Conference on Software Engineering*, pp. 455-464, Portland, USA, May 2003.

[12] L.M. Ni, Y. Liu, Y.C. Lau and A.P. Patil, "LANDMARC: Indoor Location Sensing Using Active RFID", *ACM Wireless Networks 10(6)*, pp. 701-710, Nov 2004.

[13] A. Ranganathan and R.H. Campbell, "An Infrastructure for Context-Awareness Based on First Order Logic", *Personal and Ubiquitous Computing 7*, pp. 353-364, 2003.

[14] J. Rao, S. Doraiswamy, H. Thakkar and L.S. Colby, "A Deferred Cleansing Method for RFID Data Analytics", In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pp. 175-186, Seoul, Korea, Sep 2006.

[15] R. Want, A. Hopper, V. Falcao and J. Gibbons, "The Active Badge Location System", *ACM Transactions on Information Systems 10(1)*, pp. 91-102, Jan 1992.

[16] C. Xu and S.C. Cheung, "Inconsistency Detection and Resolution for Context-Aware Middleware Support", In *Proceedings of the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 336-345, Lisbon, Portugal, Sep 2005.

[17] C. Xu, S.C. Cheung and W.K. Chan, "Incremental Consistency Checking for Pervasive Context", In *Proceedings of the 28th International Conference on Software Engineering*, pp. 292-301, Shanghai, China, May 2006.

[18] C. Xu, S.C. Cheung, W.K. Chan and C. Ye, "A Study of Resolution Strategies for Pervasive Context Inconsistency", *Technical Report HKUST-CS07-11*, Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong, China, Aug 2007.

[19] C. Xu, S.C. Cheung, W.K. Chan and C. Ye, "Consistency Constraints for Context-Aware Applications", *Technical Report HKUST-CS07-08*, Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong, China, Jul 2007.

[20] C. Xu, S.C. Cheung, W.K. Chan and C. Ye, "On Impact-Oriented Automatic Resolution of Pervasive Context Inconsistency", In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 569-572, Dubrovnik, Croatia, Sep 2007.