Heuristics-Based Strategies for Resolving Context Inconsistencies in Pervasive Computing Applications*

Chang Xu, S.C. Cheung§
Dept. of Comp. Sci. & Engg.
Hong Kong Univ. of Sci. & Tech.
{changxu, scc}@cse.ust.hk

W.K. Chan
Dept. of Comp. Sci.
City Univ. of Hong Kong
wkchan@cs.cityu.edu.hk

Chunyang Ye
Dept. of Comp. Sci. & Engg.
Hong Kong Univ. of Sci. & Tech.
cyye@cse.ust.hk

Abstract

Context-awareness allows pervasive applications to adapt to changeable computing environments. Contexts, the pieces of information that capture the characteristics of environments, are often error-prone and inconsistent due to noises. Various strategies have been proposed to enable automatic context inconsistency resolution. 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 impacts 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 inconsistency, context-awareness.

1. Introduction

Various technologies such as wireless connection, sensor networks, and Radio Frequency Identification (RFID) have enabled pervasive computing applications to interact with each other *contextually* (e.g., 'reachable by wireless connection', 'motion-detectable by a sensor', and 'identifiable by RFID signals'). *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 roar loudly in a football 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 are the 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 previous temperature readings of a room become obsolete as time passes). Human participation in manually resolving context inconsistencies is infeasible (inefficient). To resolve context inconsistencies automatically, various strategies have been proposed in the literature. Bu et al. [1] propose discarding all contexts relevant to any inconsistency. Chomicki et al. [4] suggest discarding the latest inputted event (context) if it causes any conflicting actions (inconsistencies), or to randomly discard some actions. Ranganathan et al. [13] and Insuk et al. [7] attempt to follow user preferences or policies to resolve inconsistencies in situation evaluations.

Regarding the preceding resolution strategies, we study the following research questions: *Do existing resolution strategies satisfactorily resolve inconsistent contexts for pervasive applications? Under what assumptions in pervasive computing environments can we find a better (reliable), automatic 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., sensors for location estimation or 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 the 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 impacts on context-aware applications. Section 5 discusses the experimental results and related

* 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.
§ Correspondence author.

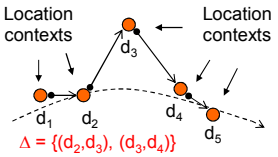

Figure 1. Five example location contexts

research issues. Finally, Section 6 presents related work and Section 7 concludes the paper.

2. Inconsistency resolution strategies

In this section, we review and compare existing automatic 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 an object's location changes with the violation of certain consistency constraints [11]. Such constraints check whether a person's location falls into a feasible area (depending on whether this person is permitted into this area), or whether the person's location change is too large so that a reasonable speed limit is violated (e.g., a person "jumped" from one floor to another floor in very short time). Suppose that in one application, the subject Peter walks steadily at an average speed of v over one period, and the application requires that *Peter's walking speed estimated from his location changes must be less than 150% of* v. Here, '150%' is selected for the 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]). The dashed curve is the actual path Peter walks, which differs from the estimated path composed of tracked locations. This is common due to the inaccuracy of existing location tracking techniques in the literature.

The preceding application requirement specifies a consistency constraint on the changes of location contexts. We first investigate location changes made by adjacent locations. In Figure 1, location d_3 seriously deviates from the actual path. 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 speed estimated from the location pair is not within the range of 150% of v). As a result, two location context inconsistencies have been identified and they are illustrated by the set Δ in Figure 1.

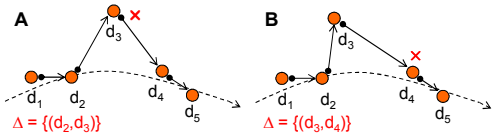

Figure 2. Drop-latest resolution strategy

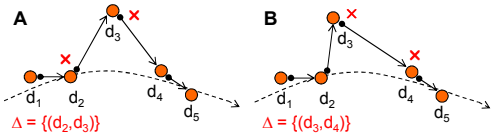

Figure 3. Drop-all resolution strategy

2.2 Drop-latest resolution strategy

In the drop-latest resolution strategy [4], the latest context leading to an inconsistency is discarded. This strategy assumes that the collection of all 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 the existing contexts.

Figure 2 shows two example scenarios. In Scenario A (the same as in Figure 1), inconsistency (d_2, d_3) is detected and context d_3 is discarded. Since d_3 has been discarded, inconsistency (d_3, d_4) cannot occur. This strategy correctly discards d_3 in Scenario A. However, in Scenario B, context d_3 is closer to d_2 than it is in Scenarios A such that the location pair (d_2, d_3) does not violate the consistency constraint. As a result, the first detected inconsistency is (d_3, d_4), and context d_4 instead of d_3 is discarded. This result is an incorrect resolution.

The drop-latest resolution strategy fails in Scenario B because even if context d_3 does not cause any inconsistency with contexts d_1 and 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 are detected) would cause the subsequent inconsistency (d_3, d_4). Then context d_4 is mistakenly discarded because this strategy always discards the latest context that causes any inconsistency. This result suggests that *determining a context immediately to be correct or incorrect based on the existence or nonexistence of a single inconsistency may not give 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 any inconsistency are incorrect and 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 both contexts d_2 and d_3 are dis-

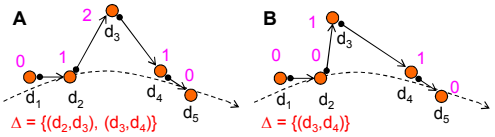

Figure 4. Counts for the two scenarios (1)

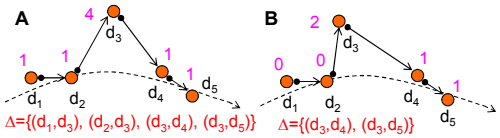

Figure 5. Counts for the two scenarios (2)

carded. Then inconsistency (d_3, d_4) cannot 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 both contexts d_3 and d_4 are discarded. This strategy also causes a context loss of d_4 that is actually correct.

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

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

3. Drop-bad resolution strategy

In this section, we present our resolution strategy.

3.1 Example revisited

Suppose that we do not resolve detected context inconsistencies immediately. Instead, we record the count for every context that participates in any inconsistency. Then we can obtain a list of counts, each of which tells how many inconsistencies a particular context has ever participated in. For illustration, we calculate the counts for Scenarios A and B in Figure 4. In Scenario A, two inconsistencies (d_2, d_3) and (d_3, d_4) are detected. Context d_3 has a count of 2 because d_3 participates in both inconsistencies and contexts d_2 and d_4 have a count of 1 because they participate in only one inconsistency. Other contexts' counts are zero because they do not participate in any inconsistency. Counts in Scenario B can be calculated in the same way.

In Scenario A, context d_3 carries the largest count (2) among all the 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, it tends to be incorrect. In Scenario B, this conjecture still holds: context d_3 carries the largest count (1), but another context d_4 also has the same count (1). Since there is only one inconsistency (d_3, d_4), we cannot dig more useful information to distinguish context d_3 from d_4. What if more inconsistency information is available?

We refine the previous consistency constraint to detect more location pairs that are not necessarily adjacent. We check whether Peter's walking speed 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 speed condition. Then, the previous context inconsistencies are still detectable, but now more context inconsistencies are 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 context inconsistencies, we observe that in both scenarios, context d_3 carries the largest count (4 and 2, respectively), and that d_3 is actually the incorrect location context we expected to discard.

This revisit makes an interesting observation that serves as the starting point of our new proposal for automatic inconsistency resolution: *one context that participates more frequently in inconsistencies is more likely to be incorrect*. We refer to our new proposal as *the drop-bad resolution strategy* and explain it in detail in the following.

3.2 Tracked context inconsistencies

Unlike existing resolution strategies that immediately resolve any context inconsistency, the drop-bad resolution strategy tolerates the inconsistency until any context participating in this inconsistency is used by any application. This strategy keeps track of all the 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 Scenario B in Figure 5 represents that two context inconsistencies have been detected but not yet resolved. The two inconsistencies are caused by context pairs (d_3, d_4) and (d_3, d_5), respectively.

Given the set of tracked context inconsistencies, we introduce a function count that returns a list of counters. Every counter records how many inconsistencies a particular 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($\Sigma$) returns $\{(d_3, 2), (d_4, 1), (d_5, 1)\}$.

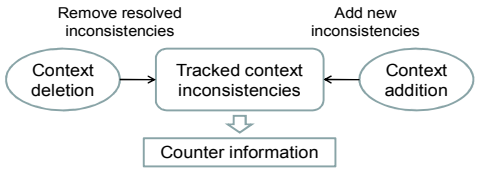

Figure 6. Tracked context inconsistencies

The set of tracked context inconsistencies is dynamic because of context changes. There are two types of context changes. One is *context addition*, which means that a new context is captured and examined by the middleware. The other is *context deletion*, which means that a previously captured context is now being used by applications and needs a decision whether it is correct. We note that context deletion only "removes" a context from the examination for its relevant inconsistencies. The context is still available until it is expired. We discuss the impacts of context changes on tracked context inconsistencies in the following (see Figure 6):

- When a context addition occurs, a new context is checked against consistency constraints. If the context causes any inconsistencies with previous contexts, Σ is added in with these inconsistencies.
- When a context deletion occurs, a previous context is to be used. Any tracked inconsistencies relevant to this context (i.e., the inconsistencies this context has participated in) need a resolution on whether this context is correct or not. These resolved context inconsistencies are removed from Σ.

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

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

For example, in 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. This strategy does not decide whether context d_3 is correct or not until subsequent contexts, d_4 and d_5, arrive. Then two more context inconsistencies (d_3, d_4) and (d_3, d_5) are detected, and the strategy thus decides that context d_3 is incorrect.

3.3 Resolution process

The resolution process of the drop-bad strategy consists of two parts (see Figure 7). The first part is for handling the case whether a new context is captured by the middleware (i.e., context addition). The second part is for handling the case that any context relevant to the tracked context inconsistencies is used by applications (i.e., context deletion).

```
Part 1: When a new context d is captured
capture a new context d
if d is irrelevant to any consistency constraint
  then d is set to consistent and made available
    to applications directly
  else d is moved to a checking buffer where any
    inconsistency relevant to d is tracked
```

```
Part 2: When a context d is used
wait for a context d in the buffer being used
if d is bad or there exists an inconsistency IC
  such that d is relevant to IC and has the
    largest count among all IC-relevant 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 is relevant to
  let d' be the context also relevant to IC' and
    having the largest count among all IC-
    relevant contexts
  if d' ≠ d then d' is set to bad
```

Figure 7. Resolution process

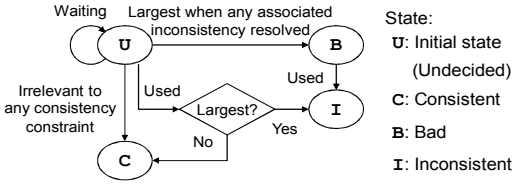

Figure 8. The life cycle of a context

We explain the resolution process using the life cycle of a context (see Figure 8). Every context has one of four states at a time: undecided, consistent, bad, or inconsistent.

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

- **Case 1: The context is used after some time.** If the count of this context is the largest among all contexts relevant to any inconsistency, the context is set to inconsistent and discarded. This is because that context is the most likely to be incorrect according to the observation stated in Section 3.1. Otherwise, if the "largest count" condition is not satisfied, the context is set to consistent because there must be a context that carries a larger count than this context, and the relevant context inconsistency can be resolved later by examining that context.
- **Case 2: This context is affected by other contexts before it is used.** Here, 'affected' means that whether this context is inconsistent would be decided ear-

lier than the time this context is used. Consider the inconsistency (d_1, d_3) in Scenario A of Figure 5. Suppose that when context d_1 is used by applications, it carries a count of 1, which is not the largest (because context d_3 carries a count of 4). Then, context d_1 is set to consistent and can be used by applications. How to handle the inconsistency (d_1, d_3)? It has been already decided that context d_1 is consistent. Then d_3, which carries the largest count (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.

Regarding the drop-bad resolution process, a question may arise: Why not discard bad contexts (e.g., context d_3 in Scenario A of Figure 5) immediately? We have the following three considerations:

i. State bad means that the context would be discarded eventually. Therefore, there is no negative effect by setting the context to bad first.
ii. Setting the context to bad respects the life cycle of the context because the context has not been used by applications yet.
iii. Setting the context to bad and then inconsistent allows the middleware to use this additional time to collect more counter information.

For example, in Scenario A in Figure 5, if we discard context d_3 immediately at the time context d_1 is used, the counter 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 strategy would be compromised. Therefore, the notion of 'bad context' tends to collect potentially more counter information for the drop-bad resolution strategy.

3.4 Generic reliability

We note that the 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.

From the observation stated in Section 3.1, 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 than that of any expected context.*

By Heuristic Rule 1, we assume that all consistency constraints are correct. In other words, there is no false report of context inconsistencies. Heuristic Rule 2 formulates our earlier observation by assuming that *all corrupted contexts* tend to participate more frequently in context inconsistencies 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 than that of any expected context.*

In the drop-bad resolution strategy, a context inconsistency is resolved by discarding some contexts relevant to this inconsistency until it vanishes. The key question is whether every discarded context is a corrupted one. The following theorem provides a theoretical foundation for this strategy:

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

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

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

We omit the proofs [18] of Theorem 1 and Theorem 2 due to page limitation. Theorem 1 and Theorem 2 explain how the drop-bad resolution strategy works reliably with generic context inconsistencies (not limited to context pairs or streaming locations). The reliability of the drop-bad resolution strategy helps address the problems (i.e., incorrect discarding and context loss problems) encountered in existing inconsistency resolution strategies (see Section 2).

One 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. This is analogous to the *test oracle problem* that makes hidden program faults hard to locate. The full treatment for a better inconsistency resolution strategy needs further study.

4. Experiments

In this section, we conduct simulation experiments to compare different strategies for context inconsistency resolution. These strategies are able to resolve context inconsistencies automatically. However, are their resolution results desirable for context-aware applications? Is the context-awareness of applications affected

during the inconsistency resolution? We study these questions in this section.

The experiments extend 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 have also designed an optimal resolution strategy (OPT-R) that is explained in Section 4.1. We conducted a total of 320 groups of experiments that were evenly distributed to the four resolution strategies with controlled error rates of contexts.

The purpose of these experiments is to measure how much a resolution strategy can affect the context-awareness of an application (e.g., adaptive behaviors 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 *the number of activated situations* as the two metrics for evaluating context-awareness. The former measures how many contexts are actually used by applications, and the latter measures how many situations are 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 the values of the two metrics under different strategies, we would be able to infer the extent of the impact on the context-awareness of applications due to a particular resolution strategy.

4.1 Experimental setup

We explain experimental settings in the following:

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 that are 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 experiences for context-aware applications [19]. To be representative, we selected only those 'popular' ones. Here, 'popular' means that the constraints or situations were designed by most participants (comprising university staff and students) in the study, and therefore they are easy to think of or widely understood. The selected constraints and situations 'cover' a large portion of all the designed constraints and situations. 'Cover' means 'identical or similar semantically'. The coverage was 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) of 10%, 20%, 30%, or 40%. These rates are based on real-life observations about RFID error rates [8][14].

Middleware. The experiments were conducted on Cabot middleware [16][17] because 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. The module could be enabled with either of the four inconsistency resolution strategies mentioned earlier.

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

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 every incorrect context. Therefore, OPT-R represents a theoretical upper bound of good strategies for correct inconsistency resolution. We set the metric values (i.e., the *number of used contexts* and *number of activated situations*) reported by OPT-R to 100% as the reference values. Then, the metric values reported by other resolution strategies (i.e., D-LAT, D-ALL, and D-BAD) are normalized to percentages against the reference values. The two percentages used in the experiments are thus called *context use rate* (ctx_use_rate) and *situation activation rate* (sit_act_rate), respectively. Intuitively, for a resolution strategy, the lower its reported percentages are, the more the applications are 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. 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 every point has been averaged over 20 groups of experiments to avoid error.

We observe that D-LAT and D-ALL are inferior to other inconsistency resolution strategies in that they have reduced the context use rate and situation activation rate by about 20-40%, compared to the optimal results reported by OPT-R. D-ALL performs the worst because it discards all contexts relevant to any incon-

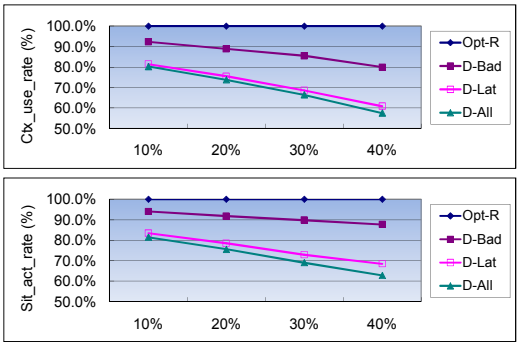

Figure 9. Resolution comparisons in application Call Forwarding

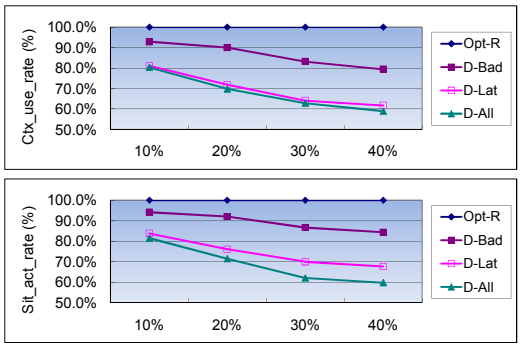

Figure 10. Resolution comparisons in application RFID data anomalies

sistency and quite a few useful contexts become 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 OPT-R. This shows that focusing on discarding corrupted contexts only cannot provide adequate support for context-aware applications. Further discussion and lessons learned are given in Section 5.

The experimental results show that simple inconsistency resolution strategies (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 error rates of contexts. D-BAD has used relationships among multiple inconsistencies (in terms of counts) to achieve better scores than other strategies. This shows that D-BAD has used better, more reasonable heuristic rules.

5. Discussions

5.1 Lessons we have learned

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

The drop-bad resolution strategy considers the counter information about multiple inconsistencies that occur within a time window. The strategy favors discarding the contexts that carry the largest counts. This implies that every context the strategy discards relates to more inconsistencies, and thus the strategy tends to resolve context inconsistencies with few discarded contexts. Nonetheless, this strategy may be trapped at a local optimum. As shown in the experiments, the drop-bad resolution strategy has brought some positive results, but still has room for improvement. For example, when the tie case comes, i.e., several contexts carrying the same maximal counts, we need to examine discarding which particular context would cause less impact on context-aware applications. This treatment would bring additional benefits to this strategy. It is a part of our ongoing work.

5.2 Further justifications

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.

We are also conducting real-life case studies to explore inconsistency resolution methods to improve the accuracy of location tracking algorithms (e.g., Landmarc [12]). Preliminary results indicate that the drop-bad resolution strategy has high scores for location survival rate (96.5%; for preventing correct locations from being discarded) and removal precision (84.7%; for guaranteeing incorrect locations to be discarded) [18]. The study also investigated an interesting question: "*how do the heuristic rules hold in practice?*" For Landmarc experiments, our results show that the first heuristic rule always holds, and that the second one (the relaxed version) holds in 91.7% of all cases. We are now on the way to further investigate what percentage is sufficient for guaranteeing satisfactory results returned from the drop-bad resolution strategy.

5.3 Several related issues

How to design correct consistency constraints? Consistency constraints specify necessary properties over useful contexts for applications. Although sufficient conditions for guaranteeing the correctness of all contexts are generally difficult, users are creative in discovering necessary properties of contexts that are relevant to what they are interested in. Our study [19] shows that users were able to identify good consistency constraints, and most of these constraints were similar to each other. Selecting these 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 experiences also apply to contexts (a special type of data and processes). Finally, the correctness of integrating consistency constraints with context-aware applications is being studied by a metamorphic approach [3]. This approach explores the ways of identifying metamorphic relations to establish consistency constraints for applications.

How to identify the time window for the drop-bad resolution strategy? It is important for this strategy to be applicable to generic applications. The time window (or the use time of a context) can be decided by the context matching time in EgoSpaces middleware [9] and Lime middleware [10] or the checking-sensitive period in Cabot middleware [16]. Besides, for RFID applications, a time window (fixed or dynamic) is usually specified for filtering out unreliable RFID data [8][14]. This time window can be used for the drop-bad resolution strategy. In the case where new contexts are used immediately, the time window of the drop-bad strategy is reduced to zero. This strategy would behave like the drop-latest strategy. As a result, the effectiveness of the drop-bad resolution is no worse than those achieved by the existing resolution strategies. As future work, the study of the impact of the time window on the effectiveness of the drop-bad resolution strategy would be an interesting and important research issue.

6. Related work

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

Researchers have addressed the context inconsistency problem and proposed similar resolution strategies. Bu et al. [1] suggest discarding all conflicting contexts except the latest one, assuming that the latest one had the largest reliability. Insuk et al. [7] and Xu et al. [16] propose to resolve context inconsistencies by following people's choices because they believe human beings can make the best decisions. However, human participation can be slow and unsuitable for dynamic environments.

Some studies are not directly related to context inconsistency but address similar problems on inconsistent or conflicting artifacts. Capra et al. [2] propose a sealed-bid auction approach to resolve 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] suggest ignoring an inputted event to avoid conflicting actions or randomly discarding some actions to resolve a conflict. Demsky et al. [5] require developers' control in specifying data's default values and the manner to generate new data during data repair. Nentwich et al. [11] present a technique to generate optional repair actions to resolve document inconsistencies, but users need to select the best choice. Ranganathan et al. [13] suggest setting up rule priorities to follow human preferences. All these approaches attempt to resolve inconsistent or conflicting artifacts, but they either require human participation or are unsuitable for pervasive computing.

Our approach follows human intuition. Discarding the contexts that participate in most context inconsistencies actually exploits redundancy to identify errors. This is similar to some pieces of work that use multiple context-detectors (e.g., multiple localization methods) to mask errors in one method by redundancy. Our approach is orthogonal to this technique in that it explores and resolves inconsistencies among different types of contexts in which errors of a single type have 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 automatic strategies for resolving context inconsistencies for pervasive computing, where distributed context sources produce 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, whereas our strategy has achieved the best results among the reviewed strategies.

Our study is mainly based on a location tracking example 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 observed a gap between the results offered by the optimal resolution strategy and those reported by the reviewed strategies. This shows rooms for improvement based on our drop-bad 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.

References

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

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

[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 (2006), pp. 677-703.

[4] J. Chomicki, J. Lobo, and S. Naqvi, "Conflict Resolution Using Logic Programming", *IEEE Trans. on Knowledge and Data Engineering5*, 1 (Jan-Feb 2003), pp. 244-249.

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

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

[7] P. Insuk, D. Lee, and S.J. Hyun, "A Dynamic Context-Conflict Management Scheme for Group-Aware Ubiquitous Computing Environments", In *Proc. the 29th Annual International Computer Software and Applications Conference*, Edinburgh, UK, Jul 2005, pp. 359-364.

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

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

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

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

[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)*, Nov 2004, pp. 701-710.

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

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

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

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

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

[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 *Proc. the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symp. on the Foundations of Software Engineering*, Dubrovnik, Croatia, Sep 2007, pp. 569-572.