Minimizing the Side Effect of Context Inconsistency Resolution for Ubiquitous Computing

Chang Xu, Xiaoxing Ma, Chun Cao, and Jian Lu

State Key Laboratory for Novel Software Technology, Nanjing University
Department of Computer Science and Technology, Nanjing University
Nanjing, Jiangsu, China
{changxu,xxm,lj}@nju.edu.cn, caochun@gmail.com

Abstract. Applications in ubiquitous computing adapt their behavior based on contexts. The adaptation can be faulty if the contexts are subject to inconsistency. Various techniques have been proposed to identify key contexts from inconsistencies. By removing these contexts, an application is expected to run with inconsistencies resolved. However, existing practice largely overlooks an application's internal requirements on using these contexts for adaptation. It may lead to unexpected side effect from inconsistency resolution. This paper studies a novel way of resolving context inconsistency with the aim of minimizing such side effect for an application. We model and analyze the side effect for rule-based ubiquitous applications, and experimentally measure and compare it for various inconsistency resolution strategies. We confirm the significance of such side effect if not controlled, and present an efficient framework to minimize it during context inconsistency resolution.

Keywords: Context inconsistency resolution, side effect, ubiquitous computing.

1 Introduction

Ubiquitous computing applications keep emerging. These applications adapt their behavior based on contexts perceived from environments. A good example is Android and iOS applications developed in recent years. They perceive environmental conditions (or *contexts*), and make continual adaptations for delivering smart services.

Unfortunately, contexts from environments usually contain uncontrolled noises. Even with data filtering [14][16], contexts may still be subject to inconsistency (or *context inconsistency*) [25][27][28]. It behaves as an application's contexts conflicting with each other by violating physical laws or application-specific rules. The application's adaptation can thus be faulty, e.g., one adaptation cancels the effect of the other, or an unexpected adaptation occurs at a wrong time or a wrong place.

Context inconsistency detection is therefore receiving attention in recent years [12] [17][21][28]. While the detection is straightforward, how to effectively resolve detected context inconsistencies is non-trivial. The contexts involved in an inconsistency are called *inconsistent contexts*. Which of them actually caused this inconsistency is usually unknown. Various techniques have proposed formulating heuristic rules, do-

A. Puiatti et al. (Eds.): MobiQuitous 2011, LNICST 104, pp. 285–297, 2012.

main knowledge, or user intentions [1][4][13][25][26] to identify key contexts from these inconsistent ones. By removing these key contexts, an application is expected to no longer suffer from context inconsistency. However, existing practice largely overlooks every application's internal requirements on using these contexts for its adaptation. Directly removing these contexts may change an application's behavior unexpectedly, especially when these contexts are identified without knowing how they are to be used in this application. We call such unexpected consequences the *side effect* of context inconsistency resolution.

As an extreme example, an application may remove all accessible contexts to resolve inconsistency. This will definitely change this application's behavior drastically. Therefore, an intuitive idea is to minimize the loss of contexts during inconsistency resolution. However, simply reducing the number of removed contexts may not work as expected, since these removed contexts may play an important role in this application. Therefore, a reasonable idea is to minimize the side effect of context inconsistency resolution according to each application's individual specification.

Our later evaluation discloses that significant side effect (over 44.0%) can result from existing inconsistency resolution techniques, and this forms a practical challenge to effective context management for ubiquitous computing. In this paper, we are interested in understanding, measuring, and minimizing such side effect according each application's individual requirements, as well as addressing challenges in doing so.

The remainder of this paper is organized as follows. Section 2 presents a running example to illustrate the side effect of context inconsistency resolution, and introduces background knowledge. Section 3 proposes our side effect measurement framework. Section 4 explains the realization of our framework and its use for measuring the side effect of context inconsistency resolution. Section 5 compares the side effect of various inconsistency resolution strategies, and shows how our framework minimizes such side effect dynamically. Finally, Section 6 presents related work and Section 7 concludes this paper.

2 Context Inconsistency Resolution and Its Side Effect

2.1 Side Effect: A Running Example

Unlike fixing traditional inconsistency in UML models [8][18] or data structures [6], resolving context inconsistency changes contexts as well as an application's behavior due to its adaptation based on the changed contexts. Since such consequences are inevitable, then *what one should protect in context inconsistency resolution?* The answer would vary with the application nature.

Consider a stock tracking application [28], in which a forklift transports stock items from the loading bay of a warehouse to its storage bay. RFID technology (RFID stands for radio frequency identification) is used to track each transported item. For safety, a rule is set up to ensure the nonexistence of missing RFID reads: *Any item detected at the loading bay should be detected again later at the storage bay*. This rule helps guard the completeness of inventory records. Context inconsistency resolution, if having to change contexts, should *protect such rules from being violated*. Besides, if context inconsistency can be resolved by *removing irrelevant contexts*, e.g.,

other forklift's location contexts, that would be even preferred. This is because useful contexts (this forklift's RFID contexts) can be thus protected from being destroyed.

This example illustrates the importance of protecting *useful properties* (e.g., safety rules) and *useful contexts* (e.g., RFID contexts). Based on this, we argue two requirements for context inconsistency resolution in ubiquitous computing:

(1) Identify key contexts from inconsistent contexts such that after removing these key contexts, the remaining contexts are inconsistency-free.
(2) If there are multiple options fulfilling the first requirement, then select the one that protect useful properties and useful contexts as many as possible.

Existing inconsistency resolution techniques have focused on the first requirement. Since they already form such "multiple options", we focus on the second requirement in this work. We use *side effect* to model how these properties and contexts are affected by inconsistency resolution, and propose to minimize it at runtime along with inconsistency resolution. To facilitate our discussions, we brief background knowledge about context inconsistency resolution below.

2.2 Context Inconsistency Resolution Techniques and Strategies

Context inconsistency stems from several reasons. A major one is noisy data. For example, RFID read rate can drop to 60-70% in real-life deployment [14]; GPS errors are often tens of meters; for GSM cellphone network, field tests can result in errors of 187-287 meters [24]. People have proposed various filter and threshold techniques to smooth these noisy data [14], or measured them probabilistically with uncertainty levels [16]. Still, data-level techniques cannot completely prevent a ubiquitous computing application from suffering context inconsistency.

Context inconsistency may also come from the failure of synchronizing all contexts [22] or the absence of a global consistency of all environmental conditions [19]. Due to such complexity, context inconsistency is receiving growing attention in recent years. There are roughly three categories of existing work on addressing context inconsistency or inconsistent contexts.

One category takes an *application-specific* approach based on context type or application nature. For example, Deshpande et al. [7] proposed smoothing location contexts by interpolation techniques; Jeffery et al. [14] proposed adjusting sensing window size to retrieve missing RFID contexts. These pieces of work use geometry knowledge or probabilistic models to preprocess contexts, e.g., making location contexts form an expected curve or RFID contexts statistically follow a distribution. They are not generally applicable to other types of contexts or applications.

The second category targets at more applications by *heuristic rules*. These rules are formulated by domain experts or from empirical experiments. They suggest key contexts for removal in order to make resulting contexts inconsistency-free. Such rules can be conservative by removing all inconsistent contexts [1], or optimistic by denying the latest contexts in inconsistent ones from being accessible to applications [4]. The selection of such key contexts can also be random [4] or follow some criterion, e.g., minimizing the number of all removed contexts [26].

The third category follows user's *preferences or priorities*. They can be statically decided in advance [13][20][25], or calls for user's participation at runtime to best fit

user's intentions dynamically [23].

While these techniques vary in their formulation and effectiveness, their consequences on resulting contexts are similar, behaving as some key contexts are removed and remaining ones are accessible to applications. Therefore, we classify these techniques according to their consequences on resulting contexts. Our classification includes the following four strategies that cover the aforementioned techniques (these four strategies are to be used in our later evaluation):

(1) ALL: Removing all inconsistent contexts;
(2) LATEST: Removing the latest context (a representative of those fixed criteria);
(3) RANDOM: Removing a random context (a representative of random criteria);
(4) FEWER: Minimizing the number of all removed contexts.

3 Side Effect Measurement Framework

Side effect measurement concerns how contexts are being used in an application in order to calculate how useful properties and contexts are affected. In the following, we first explain the concept of context, application specification, and side effect, and then present a framework to measure the side effect at runtime.

3.1 Context and Application Specification

In ubiquitous computing, new contexts characterizing environmental conditions keep emerging. Usually only a subset of recent contexts is accessible to applications. They are called *available contexts*.

Available contexts are used in an application according to this application's design logic. A large body of ubiquitous computing applications has their logics formulated by *adaptation rules*, which specify what to do under what conditions [11][20][22]. The set of such rules is called *application specification*. An application specification includes information about interesting contexts and the conditions under which adaptation should take place. They are important clues about useful contexts and properties concerned by this application.

Such rule-based applications are being widely used and supported by many context middleware or frameworks [20][25], and they are our focus in the paper.

3.2 Context Use and Side Effect

If available contexts are already inconsistency-free, they are safe for using by an application specification. The process of evaluating contexts according to an application specification is called *context evaluation*. For example, the aforementioned stock tracking application checks whether available contexts indicate the forklift has arrived at the storage bay so that it can unload transported items. This checking process is an example of context evaluation. Let available contexts be A, application specification be S, and context evaluation be $\otimes_E$. Fig. 1 abstracts this context use scenario.

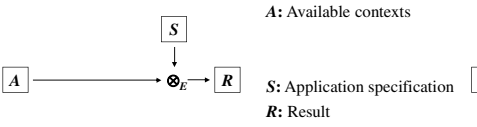

Fig. 1. Context use in an application (*without* context inconsistency resolution)

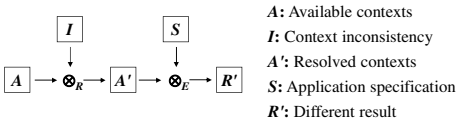

Fig. 2. Context use in an application (*with* context inconsistency resolution)

If available contexts contain any inconsistency, then the inconsistency should be resolved. Fig. 2 illustrates how available contexts A are first processed for resolving inconsistency I. Then resolved contexts A' are used for context evaluation by application specification S. $\otimes_R$ represents the process of context inconsistency resolution.

We note that R in Fig. 1 and R' in Fig. 2 are two different context evaluation results. Their comparison discloses the side effect of context inconsistency resolution. As different resolution strategies correspond to different processes of context inconsistency resolution, n strategies can lead to n different evaluation results R_1, R_2, …, R_n. Then the side effect can be measured by comparing these R_1, R_2, …, R_n to a base value R_{base}. In practice, R_{base} can be set to R, which represents the context evaluation result without any inconsistency resolution. If one finally selects a resolution strategy k ($1 \le k \le n$) that *minimizes* the difference between R_k and R, it implies the effort of: (1) first resolving context inconsistency, and then (2) making the application behave as *similar* as no context inconsistency resolution occurred. This effort is reasonable and makes sense in most cases.

The comparison between R_k and R also concerns what metric to measure. It is decided by what one plans to protect in context inconsistency resolution. To protect useful contexts, the *number or types of contexts* that are referred to in evaluation result R_k can be calculated as the side effect metric for comparison. If an application specification also contains useful properties like safety rules, the *number of instances satisfying these properties* can also be calculated from R_k as the side effect metric.

3.3 Side Effect Measurement

Side effect measurement seems straightforward but actually not. The major issue is the complexity caused by multiple context evaluations. Given n resolution strategies to compare, context evaluation has to be conducted n times, with n *different* sets of resolved contexts A_k ($1 \le k \le n$). Fig. 3 (left) illustrates the whole picture, and the part in the dashed rectangle is what one has to complete in order to compare n evaluation results. This part has to be completed *efficiently* as it works at runtime.

We address this challenge using our incremental measurement idea. From Fig. 3 (left), n context evaluations differ only at n inputted sets of resolved contexts A_k ($1 \le k \le n$). These sets of resolved contexts come from the same available contexts A and same context inconsistencies I, but with a different resolution process $\otimes_{R_k}$ ($1 \le k \le n$). If one detects context inconsistency in short periods or incrementally [28], the number of context inconsistencies in each period would be a *small constant*. Then the n sets of resolved contexts A_1, A_2, …, A_n would be similar to each other. This motivates us to share the computation across n context evaluations with little cost.

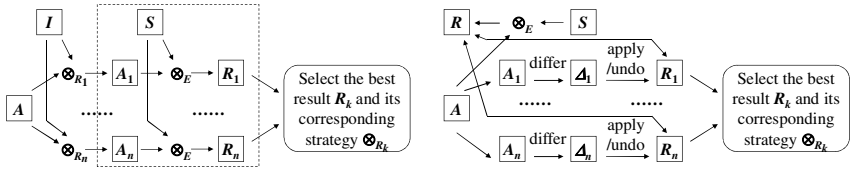

Fig. 3. Side effect measurement framework (left: direct; right: incremental)

We brief our incremental side effect measurement idea below for Fig. 3 (right):

(1) Conduct context evaluation on available contexts A and application specification S, and obtain result R for later reference.
(2) For each resolution process $\otimes_{Rk}$, calculate the difference Δ_k between resolved contexts A_k and available contexts A. Then apply Δ_k to reference result R to obtain updated result R_k.
(3) Undo Δ_k to restore R_k back to R.
(4) Repeat Steps (2)-(3) for all n resolution processes, and finally select the best one that minimizes the given side effect metric (subject to user's choice).

The novelty of this idea is that one does not have to conduct n *complete* context evaluations. Instead, one only needs to conduct it *once*, and later *update* it n times for different results. The benefits include reduced computation time (almost down to $1/n$ time) as well as reduced space cost (no need to store n intermediate evaluation results). However, the key to success is how one can *efficiently* update reference result R to obtain required result R_k and later restore it back to R. This concerns underlying data structures and operations, and we present one realization for them below.

4 Realization of the Framework

In this section, we discuss the realization of our incremental side effect measurement framework. We start with basic blocks of context, pattern, and rule in an application specification, and explain key data structures for supporting efficient context evaluation and its result update and restoration.

4.1 Context, Pattern, and Rule

Context. Context is represented as a tuple with multiple fields, each of which is a name-value pair. This is comparable to many pieces of existing work on context modeling [13][20][21] in order to be representative. For example, an RFID context can be represented as ((type, "RFID"), (subject, "tag 0327"), (predicate, "detected by"), (object, "reader a"), (time, "10:20:05am")).

Pattern. Pattern is used to select interesting contexts satisfying predefined conditions in this pattern. For example, pattern ((type, "RFID"), (subject, "tag 0327")) select all RFID contexts about tag 0327, i.e., this tag is being continually tracked by this pattern. When context c satisfies the conditions in pattern P, we say that c matches P, represented as $c \in P$.

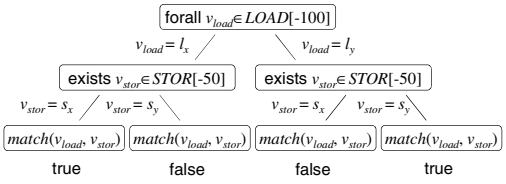

Fig. 4. Context evaluation tree for the safety rule in the stocking tracking application

Rule. A modeling language is used to specify rules in an application specification. It is based on first-order logic with timing constraints. Its expressive power is comparable to existing work on specifying adaptation rules [16][20][22][25]:

f ::= forall $v \in P[t]$ (f) | exists $v \in P[t]$ (f) | (f) and (f) | (f) or (f) | (f) implies (f) | not (f) | $bfunc(v, \ldots, v)$.

The above syntax follows their traditional first-order logic interpretations. $P[t]$ refers to a pattern P that selects contexts restricted by a period of t. Terminal *bfunc* refers to any application-specific function that returns a Boolean value.

An application specification can contain multiple rules, each of which is constructed by recursively using the above syntax. For example, the following rule specifies the aforementioned safety rule in the stock tracking application:

forall $v_{load} \in LOAD$[-100] (exists $v_{stor} \in STOR$[-50] ($match(v_{load}, v_{stor})$)).

This rule specifies that each RFID context perceived at the loading bay (in the past 100s) should be able to find its matched RFID context at the storage bay later within a specified period (in the past 50s), implying the nonexistence of missing RFID reads.

4.2 Context Evaluation Tree

We use *context evaluation tree* (CET) to represent how context evaluation is conducted for a rule in the application specification. Each rule owns such a tree. By this tree, we can: (1) efficiently update it for a new context evaluation, and (2) measure the aforementioned side effect metrics for useful contexts and properties.

Suppose $LOAD$[-100] = $\{l_x, l_y\}$ and $STOR$[-50] = $\{s_x, s_y\}$. Then the CET of the safety rule can be constructed as Fig. 4 shows. The construction naturally follows the rule's syntactic hierarchy. From this CET, the *number of useful contexts* can be easily calculated (4), as a total of four contexts have participated in the context evaluation of this rule. Besides, the *number of instances satisfying the safety rule* can also be derived (2), as two pairs of contexts ((l_x, s_x) and (l_y, s_y)) satisfy the *match* function. We next explain how a CET can be efficiently updated for computation reuse (reuse for n context evaluations in comparing n resolution strategies).

4.3 CET Update

We first explain efficient construction of a CET. As a CET is based on available contexts A and application specification S, it can be seen as the result of applying a series of *context changes* (constituting A) to an empty CET (following a rule's syntactic hierarchy in S). These context changes include the following two types:

New Context. When a new context emerges from environment, a relevant CET can be *incrementally* updated to incorporate this context. For example, if $STOR$[-50] has a

new context s_z, then two "exists $v_{stor} \in STOR$[-50]" nodes should be attached with a new branch corresponding to this new context s_z as existing contexts s_x, s_y.

Expired Context. If a previous context expires due to its timing constraint, a relevant CET can also be *incrementally* updated to incorporate this change. For example, if *LOAD*[-100]'s previous context l_x expires, then the branch corresponding to context l_x (with three nodes) should be removed from the tree.

Therefore, the reference result R required in Step (1) of the side effect measurement framework can be efficiently calculated without reconstructing whole CETs when any context change occurs. For the side effect comparison in Steps (2)-(3), we note that R should also be efficiently updated to obtain R_k ($1 \le k \le n$) for n resolution strategies. This can be done by updating relevant CETs to incorporate the following two change types from resolution strategies:

Removed Context. When a resolution strategy needs to remove a context for resolving inconsistency in Step (2), we *temporarily* remove branches corresponding to this context from relevant CETs (similar to "expired context"). However, these branches are kept in a buffer. Later when the framework needs to undo this resolution strategy in Step (3), we restore these branches to their original places.

Added Context. We also support some rare resolution strategies that add new contexts to resolve inconsistency. This can be done by *temporarily* adding corresponding branches to relevant CETs (similar to "new context"). These branches are removed later to undo the effect of a resolution strategy.

Therefore, applying and undoing a resolution strategy can also be *incrementally* realized without reconstructing whole CETs.

4.4 Efficiency Analysis

Due to space limitation, we only brief our efficiency analysis results below. Let the height of a CET be h and its node number be d. We have:

(1) Handling a "new context"/"added context" change takes O(h) – O(d) time;
(2) Handling an "expired context"/"removed context" change takes O(h) time;
(3) Undoing an "added context"/"removed context" change takes O(1) time.

In our side effect measurement framework, each resolution strategy takes a limited number of steps to resolve context inconsistency detected in small periods. Let the number be restricted by a constant l (usually 0.3-0.5 in our experiments). We have:

Step (2): Applying n resolution strategies takes O($n \cdot l \cdot h$) – O($n \cdot l \cdot d$) time;
Step (3): Undoing n resolution strategies takes O($n \cdot l \cdot h$) time.

Consider that l is a small constant, h is fixed (also a constant), and almost all resolution strategies remove selected inconsistent contexts only (factor d can thus be removed). Then Steps (2) and (3) take O(n) time, which implies that *the whole framework takes O(1) time to measure the side effect for each resolution strategy* (Step (1) is incrementally conducted as explained and can be merged into later steps). This guarantees the framework to be *efficient* and *scalable* at runtime.

A buffer needs to be maintained for keeping temporarily removed branches, but at the same time, the same size of space is released from relevant CETs. Therefore, there is no extra space cost in addition to what is required by CETs themselves.

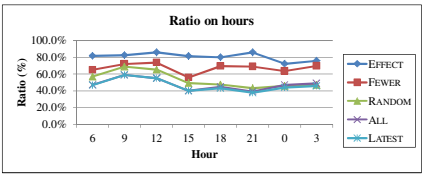

Fig. 5. Side effect comparison (hour-based)

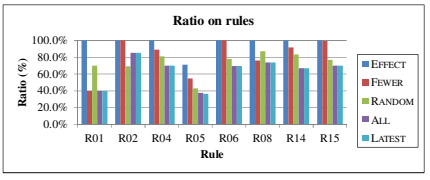

Fig. 6. Side effect comparison (rule-based)

5 Experimentation

We use an open-source context simulator Siafu (http://siafusimulator.sourceforge.net/) to measure and compare the side effect of different context inconsistency resolution strategies. Siafu loads the data of a realistic town of Leimen in Germany. It ran for a continuous 24-hour day and generated a total of 288,000 contexts. These contexts contain a controlled error rate of 15% and were checked against 12 consistency constraints for inconsistency. The contexts and detected inconsistencies were used for our experiments.

To measure the side effect of context inconsistency resolution on ubiquitous applications, we studied Active Campus [9], Gaia [20][21], Socam [10], CARISMA [2], Egospaces [15], and Runes [5]. From them, we formulated 16 rules as the application specification for our experiments. The specification contains varying requirements of using contexts for adaptation. Its mixed nature helps alleviate possible bias caused by any single application.

The experiments were conducted on a machine with Intel Core 2 Duo 2.13GHz CPU and 1GB RAM. Software includes Windows XP Professional SP3 and Oracle/Sun JRE 1.6. The side effect is measured by the ratio between R_k and R on a metric that combines both the number of useful contexts and number of instances satisfying useful properties, with an equal weight for experimental purposes.

Fig. 5 compares the side effect of five resolution strategies based on different *hours* (starting at 6am). These strategies include FEWER, RANDOM, ALL, and LATEST discussed earlier in Section 2. A new strategy is EFFECT, which compares the above four strategies at runtime and always selects the one that minimizes the side effect (by highest ratios). From Fig. 5, all strategies caused side effect. RANDOM, ALL, and LATEST are most severe with a ratio of 38.0-68.8% (31.2-62.0% side effect). FEWER is better with a ratio of 55.8-73.5% (26.5-44.2% side effect). EFFECT's ratio is highest (72.1-85.7%), implying the least side effect (14.3-27.9%).

Fig. 6 compares the side effect based on different *rules*. Among 16 rules, eight of them (R01-02, R04-06, R08, R14-15) are affected by inconsistent contexts. RANDOM, ALL, and LATEST are most severe with a ratio of 36.6-85.5% (14.5-63.4% side effect). FEWER is better with a ratio of 40.0-100.0% (0.0-60.0% side effect). EFFECT's ratio is still highest (71.1-100.0%), meaning the best control on the side effect (0.0-28.9%).

On average, RANDOM, ALL, and LATEST have a ratio of 56.0%, 49.4%, and 48.5%, which shows *significant* side effect (over 44.0%) resulted from context inconsistency

resolution. FEWER behaves better: 67.5% (32.5% side effect), which results from its nature that tries to minimize the number of all removed contexts in resolving inconsistency. This helps protect applications from losing useful contexts and properties to some degree. By dynamically selecting the strategy that minimizes the side effect at runtime, EFFECT improves the average ratio to 81.4% (18.6% side effect). We note that zero side effect may not be possible as inconsistency resolution inevitably changes contexts and an application's behavior. Still, EFFECT makes such attempt and controls context inconsistency resolution with the *least* side effect on applications.

We owe this ability to the framework's runtime efficiency and scalability (negligible experimental time: totally several seconds for all 24-hour contexts). Otherwise, it would have failed to measure and compare the side effect dynamically.

6 Related Work

Ubiquitous computing and context-awareness are receiving increasing attention. Various application frameworks [11][15] and middleware infrastructures [2][10][20][25] have been proposed to support the development of context-aware ubiquitous applications. These applications may be subject to context inconsistency at runtime, and therefore call for efforts to address context inconsistency.

People proposed various techniques to detect context inconsistency efficiently by reusing previous checking results [28], or asynchronously by identifying distributed inconsistency-triggering events that occur concurrently [12]. Detected context inconsistencies can be modeled or analyzed in different forms like application exceptions against normal work routines [17], or semantic conflicts when different terminologies or semantics are mixed [21].

Context inconsistency resolution work like [14][16] focuses on filtering raw contextual data probabilistically and marking remaining ones with uncertainty levels. While this gives useful hints on how likely they are correct, applications still face problems when selecting useful contexts from them without knowing possible consequences by doing so. Domain knowledge or user observations can be formulated as heuristic rules or user preferences [1][4][13][20][23][25][26], but they help little on this issue. As disclosed by our analysis and experimental results, these pieces of work suffer from *uncontrolled* side effect that impairs useful contexts and properties specific to certain applications.

A recent piece of work [3] proposed resolving context inconsistency based on application logics. This shares some observations as ours. However, this work assumes the availability of effect function for each action in an application, and requires error recovery plans to compensate what has been caused by context inconsistency.

Our work is based on our earlier efforts for context inconsistency detection [28] and manual resolution with fixed policies [25]. We later extended the work with heuristic rules to automatically resolve context inconsistency [26]. We identified negative consequences caused by context inconsistency resolution, and experimentally measured them for two applications [27]. Based on our earlier preliminary efforts, in this

paper we formulated the side effect issue in context inconsistency resolution, and presented an efficient side effect measurement framework. Our incremental measurement technique enables *runtime* side effect calculation and comparison, allowing the selection of the best resolution strategy with the *least* side effect on applications.

7 Conclusion

The study in this paper measures the *significant side effect* caused by context inconsistency resolution on ubiquitous applications. It shows that side effect can be a *new criterion* for evaluating various inconsistency resolution techniques in addition to their original objectives. It can be further explored what is the most suitable base to which the side effect should be compared, and how the most suitable measurement metric can be selected. The answers should be application-specific. Our framework provides *a systematic way to measure such side effect*, as well as *an efficient realization* to compare different resolution strategies at runtime. This suits for applications whose requirements of using contexts are not static and can evolve dynamically (i.e., context-aware). We are now working on realistic experiments and automated context repair techniques towards better quality guarantee for ubiquitous applications.

Acknowledgments. This research was partially funded by National Science Foundation (grants 60736015, 61021062, 61100038) and 863 program (2011AA010103) of China. Chang Xu was also partially supported by Program for New Century Excellent Talents in University (NCET-10-0486).

References

[1] Bu, Y., Gu, T., Tao, X., Li, J., Chen, S., Lu, J.: Managing Quality of Context in Pervasive Computing. In: 6th Inter. Conf. on Quality Software, Beijing, China, pp. 193–200 (October 2006)

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

[3] Chen, C., Ye, C., Jacobsen, H.: Hybrid Context Inconsistency Resolution for Context-aware Services. In: IEEE Inter. Conf. on Pervasive Computing and Communications, Seattle, Washington, USA, pp. 10–19 (March 2011)

[4] Chomicki, J., Lobo, J., Naqvi, S.: Conflict Resolution Using Logic Programming. IEEE Trans. on Knowledge and Data Engineering 15(1), 244–249 (2003)

[5] Costa, P., et al.: The RUNES Middleware for Networked Embedded Systems and Its Application in a Disaster Management Scenario. In: 5th Annual IEEE Inter. Conf. on Pervasive Computing and Communications, White Plains, NY, USA, pp. 69–78 (March 2007)

[6] Demsky, B., Rinard, M.C.: Goal-directed Reasoning for Specification-based Data Structure Repair. IEEE Trans. on Software Engineering 32(12), 931–951 (2006)

[7] Deshpande, A., Guestrin, C., Madden, S.R.: Using Probabilistic Models for Data Management in Acquisitional Environments. In: 2nd Biennial Conf. on Innovative Data Systems Research, Asilomar, California, USA, Article 26, pp. 1–13 (January 2005)
[8] Egyed, A.: Fixing Inconsistencies in UML Design Models. In: 29th Inter. Conf. on Software Engineering, Minneapolis, MN, USA, pp. 292–301 (May 2007)
[9] Griswold, W.G., Boyer, R., Brown, S.W., Tan, M.T.: A Component Architecture for an Extensible, Highly Integrated Context-aware Computing Infrastructure. In: 25th Inter. Conf. on Software Engineering, Portland, USA, pp. 363–372 (May 2003)
[10] Gu, T., Pung, H.K., Zhang, D.Q.: Toward an OSGi-based Infrastructure for Context-aware Applications. In: 2nd IEEE Inter. Conf. on Pervasive Computing and Communications, Orlando, Florida, USA, pp. 66–74 (March 2004)
[11] Henricksen, K., Indulska, J.: A Software Engineering Framework for Context-aware Pervasive Computing. In: 2nd IEEE Conf. on Pervasive Computing and Communications, Orlando, Florida, USA, pp. 77–86 (March 2004)
[12] Huang, Y., Ma, X., Cao, J., Tao, X., Lu, J.: Concurrent Event Detection for Asynchronous Consistency Checking of Pervasive Context. In: 7th Annual IEEE Inter. Conf. on Pervasive Computing and Communications, Galveston, Texas, USA, pp. 131–139 (March 2009)
[13] Insuk, P., Lee, D., Hyun, S.J.: A Dynamic Context-conflict Management Scheme for Group-aware Ubiquitous Computing Environments. In: 29th Annual Inter. Computer Software and Applications Conf., Edinburgh, UK, pp. 359–364 (July 2005)
[14] Jeffery, S.R., Garofalakis, M., Frankin, M.J.: Adaptive Cleaning for RFID Data Streams. In: 32nd Inter. Conf. on Very Large Data Bases, Seoul, Korea, pp. 163–174 (September 2006)
[15] Julien, C., Roman, G.C.: EgoSpaces: Facilitating Rapid Development of Context-aware Mobile Applications. IEEE Trans. on Software Engineering 32(5), 281–298 (2006)
[16] Khoussainova, N., Balazinska, M., Suciu, D.: Towards Correcting Input Data Errors Probabilistically Using Integrity Constraints. In: 5th Inter. ACM Workshop on Data Engineering for Wireless and Mobile Access, Chicago, Illinois, USA, pp. 43–50 (June 2006)
[17] Kulkarni, D., Tripathi, A.: A Framework for Programming Robust Context-aware Applications. IEEE Trans. on Software Engineering 36(2), 184–197 (2010)
[18] Nentwich, C., Emmerich, W., Finkelstein, A.: Consistency Management with Repair Actions. In: 25th Inter. Conf. on Software Engineering, Portland, USA, pp. 455–464 (May 2003)
[19] Rajamani, V., Julien, C.: Blurring Snapshots: Temporal Inference of Missing and Uncertain Data. In: 8th Annual IEEE Inter. Conf. on Pervasive Computing and Communications, Mannheim, Germany, pp. 40–50 (March-April 2010)
[20] Ranganathan, A., Campbell, R.H.: An Infrastructure for Context-awareness Based on First Order Logic. Personal and Ubiquitous Computing 7, 353–364 (2003)
[21] Ranganathan, A., Campbell, R.H., Ravi, A., Mahajan, A.: ConChat: A Context-aware Chat Program. IEEE Pervasive Computing 1(3), 51–57 (2002)
[22] Sama, M., Elbaum, S., Raimondi, F., Rosenblum, D.S., Wang, Z.: Context-aware Adaptive Applications: Fault Patterns and Their Automated Identification. IEEE Trans. on Software Engineering 36(5), 644–661 (2010)
[23] Shin, C., Dey, A.K., Woo, W.: Mixed-initiative Conflict Resolution for Context-aware Applications. In: 10th Inter. Conf. on Ubiquitous Computing, Seoul, Korea, pp. 262–271 (2008)

[24] Wu, Z.L., Li, C.H., Ng, J.K.Y., Leung, K.R.P.H.: Location Estimation via Support Vector Regression. IEEE Trans. on Mobile Computing 6(3), 311–321 (2007)

[25] Xu, C., Cheung, S.C.: Inconsistency Detection and Resolution for Context-aware Middleware Support. In: Joint 10th European Software Engineering Conf. and 13th ACM SIGSOFT Symp. on the Foundations of Software Engineering, Lisbon, Portugal, pp. 336–345 (September 2005)

[26] Xu, C., Cheung, S.C., Chan, W.K., Ye, C.: Heuristics-based Strategies for Resolving Context Inconsistencies in Pervasive Computing Applications. In: 28th Inter. Conf. on Distributed Computing Systems, Beijing, China, pp. 713–721 (June 2008)

[27] Xu, C., Cheung, S.C., Chan, W.K., Ye, C.: On Impact-oriented Automatic Resolution of Pervasive Context Inconsistency. In: 6th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. on the Foundations of Software Engineering, Dubrovnik, Croatia, pp. 569–572 (September 2007)

[28] Xu, C., Cheung, S.C., Chan, W.K., Ye, C.: Partial Constraint Checking for Context Consistency in Pervasive Computing. ACM Trans. on Software Engineering and Methodology 19(3), Article 9, 1–61 (2010)