To appear in *Proceedings of the 28th International Conference on Software Engineering (ICSE 2006), Shanghai, China 20-28, 2006.*

Incremental Consistency Checking for Pervasive Context*

Chang Xu
Department of Computer Science
Hong Kong University of Sci. & Tech.
Kowloon, Hong Kong, China
changxu@cs.ust.hk

S.C. Cheung§
Department of Computer Science
Hong Kong University of Sci. & Tech.
Kowloon, Hong Kong, China
scc@cs.ust.hk

W.K. Chan
Department of Computer Science
Hong Kong University of Sci. & Tech.
Kowloon, Hong Kong, China
wkchan@cs.ust.hk

ABSTRACT

Applications in pervasive computing are typically required to interact seamlessly with their changing environments. To provide users with smart computational services, these applications must be aware of incessant context changes in their environments and adjust their behaviors accordingly. As these environments are highly dynamic and noisy, context changes thus acquired could be obsolete, corrupted or inaccurate. This gives rise to the problem of context inconsistency, which must be timely detected in order to prevent applications from behaving anomalously. In this paper, we propose a formal model of incremental consistency checking for pervasive contexts. Based on this model, we further propose an efficient checking algorithm to detect inconsistent contexts. The performance of the algorithm and its advantages over conventional checking techniques are evaluated experimentally using *Cabot* middleware.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification – *Validation*

General Terms

Algorithms, Performance, Theory, Verification

Keywords

Pervasive Computing, Context Management, Incremental Consistency Checking

1. INTRODUCTION

Pervasive computing applications are often context-aware, using various kinds of contexts such as user posture, location and time to adapt to their environments. For example, a smart phone would vibrate rather than beep seamlessly in a concert hall, but would beep loudly during a football match. To enable an application to behave smartly, valid contexts should be available. For example, it would be embarrassing if the smart phone roared during the most important moment of a wedding ceremony if the context mistakenly dictates that the environment is a football match. This poses a natural requirement on *consistent contexts* [24].

ICSE'06, May 20–28, 2006, Shanghai, China.

A *context* of a computation task in pervasive computing refers to an attribute of the situation in which the task takes place. Essentially, a context is a kind of data. It differs from data in conventional databases in the sense that data in a conventional database are integral and consistent during a database transaction. On the other hand, it is difficult for contexts to be precise, integral and non-redundant. For example, in a highly dynamic environment, contexts are easily obsolete; context reasoning may conclude inaccurate results if its relying contexts have partially become obsolete during the reasoning computation. A detailed analysis of the reasons on inevitably imperfect contexts for pervasive computing can be found in [24].

As a result, we need to deal with inconsistent contexts in pervasive computing. To improve the consistency of the computing environment for its embedding context-aware applications, it is desirable to timely identify inconsistent contexts and prevent applications from using them. Still, every application may cumber to handle similar consistency issues individually. Appropriate common abstraction layers, such as context middleware or frameworks [10] [11] [12], to provide context querying or subscription services of a high degree of consistency are attractive. One promising approach is to check consistency constraints [16] that meet application requirements at runtime to avoid inconsistent contexts being disseminated to applications.

We give the following example: *Call Forwarding* [23] is a location-aware application, which aims at forwarding incoming calls to the target callees with phones nearby. *Call Forwarding* assumes phone receptionists knowing the callees' current locations. Generally, raw sensory data are collected by the underlying *Active Badge System* which is built on the infrared technology, and these raw data are converted into location contexts by certain algorithms. To warrant accurate location contexts, consistency constraints (e.g. "nobody could be in two different places at the same time") have to be specified and checked. Any violation of such constraints indicates the presence of inconsistent location contexts, which must be detected in time to prevent the application from taking inappropriate action. In this example, the goal of acquiring precise location contexts becomes the basis to reject inconsistent location contexts that violate the pre-specified consistency constraints.

* This research is supported by a grant from the Research Grants Council of Hong Kong (Project No. HKUST6167/04E).

§ All correspondence should be addressed to S.C. Cheung at the Department of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong. Tel: (+852) 23587016. Fax: (+852) 23581477. Email: scc@cs.ust.hk.

The checking of consistency constraints, or *consistency management*, has been recognized as an important issue by software engineering and programming communities [14]. Generally, consistency constraints are expressed as rules and checked over interesting content (e.g., documents and programs) such that any violation of rules, called *inconsistency*, can be detected. The inconsistency is presented to users or checking systems for further actions such as carrying out repairs.

Various studies have been conducted on consistency checking [2] [3] [14] [19]. Their research attention is paid mainly to interesting content changes and checking is performed eagerly or lazily to identify if any change has violated predefined rules. We take the *xlinkit* framework [14] from them for discussion. *xlinkit* is a well recognized tool for consistency checking of distributed *XML* documents. In *xlinkit*, consistency constraints are expressed as rules in first order logic (*FOL*) formulae. For every rule, their approach employs *xpath* selectors to retrieve interesting content from *XML* documents. The approach is intuitive in the sense that each time the entire set of *XML* documents is checked against all rules. We term this type of checking *all-document checking*.

However, all-document checking is computationally expensive when applied to large datasets that require frequent checking. *xlinkit*'s improved model [16] can identify the rules whose targeted content is potentially affected by a given change. Only these affected rules need to be rechecked against the entire dataset upon the change. Since it is a rule-level checking approach, we term it *rule-based incremental checking*.

Rule-based incremental checking is attractive when there are many rules but each change in the content only affects a few of them. Still, the approach suffers from two limitations. First, the entire formula of an affected rule should be rechecked even though a small sub-formula is affected by the changes. In practice, a rule typically contains universal or existential sub-formulae in which variable assignments are subject to changes [16]. If the granularity of checking can be confined to sub-formulae, it saves many rechecking efforts. Second, each affected rule has to be rechecked against the entire dataset or documents even for small changes. This is because the last retrieved content from *xpath* selectors was not stored, and thus could not be reused in the subsequent checking. The above two limitations present a major challenge when we apply the rule-based incremental checking approach to dynamic pervasive computing environments, in which applications require timely responses to context changes and typically context changes rapidly but each change is usually marginal.

We envisage using a finer checking granularity and the consistency checking results of previous rounds to achieve efficient consistency checking for pervasive computing. Our proposal addresses the following two challenges: (1) Can the checking granularity be reduced from rules to sub-formulae? That is, for a given rule that needs rechecking, we only check those sub-formulae directly affected by context changes and use the last checking results of other sub-formulae to get the updated final result of the rule. (2) Can stateless *xpath* selectors be replaced with some stateful context retrieval mechanism? We maintain the latest retrieved contexts and update them at the time when any interesting context change occurs. Thus, we do not have to re-parse the whole context history each time. Our experiments report that our technique can achieve more than a five-fold performance improvement.

Although various studies such as [2] [3] [14] [19] have addressed the second challenge, the first has not yet been examined. In this paper, we address the two challenges by presenting a novel consistency checking technique, called *formula-based incremental checking*. In particular, our contributions include:

- The use of context patterns combined with *FOL* formulae in expressing consistency constraints on contexts;
- The formulation of the Boolean value semantics and link generation semantics for incremental consistency checking; and
- The proposal of an efficient and incremental algorithm to detect context inconsistency for pervasive computing.

The remainder of the paper is organized as follows. Section 2 introduces recent related work. Sections 3 and 4 introduce preliminary concepts on context modeling and consistency checking, respectively. Section 5 discusses three closely related issues: incremental Boolean value evaluation, incremental link generation and stateful context patterns. They are followed by the algorithm implementation in Section 6 and a group of comparison experiments in Section 7. The last section concludes the paper.

2. RELATED WORK

Context consistency management has not been adequately studied in the existing literature. A few studies on context-awareness are concerned with either the frameworks that support context abstraction or data structures that support context queries [9] [10] [21], but the detection of inconsistent contexts is rarely discussed. Some projects, including *Gaia* [20], *Aura* [22] and *EasyLiving* [4], have been proposed to provide middleware support for pervasive computing. They focus on the organization of and the collaboration among computing devices and services. Other infrastructures are mostly concerned with the support of context processing, reasoning and programming. An earlier representative work is *Context Toolkit* presented in [6]. It assists developers by providing abstract components to capture and process context data from sensors. *Context Toolkit* falls short of supporting highly integrated applications, so Hence Griswold et al. in [8] propose to apply a hybrid mediator-observer pattern in the system architecture. To facilitate context-aware computing, Henricksen et al. in [10] present a multi-layer framework that supports both conditional selection and invocation of program components. Ranganathan et al. in [18] discuss how to resolve potential semantic contradictions in contexts by reasoning based on first-order predicate calculus and Boolean algebra. These works have conducted initial research on context programming, certainty representation and uncertainty reasoning, but inadequate attention has been paid to comprehensive context consistency management in pervasive computing.

Pervasive computing also shares many observations with artificial intelligence (*AI*), active databases and software engineering disciplines. In *AI*, expert systems to support strategy formulation and decision-making are common. Many efforts have been made on the evidence aggregation problem so that the systems can develop strategies over contradicting rules [24]. In active databases, advanced event detection techniques are proposed for triggering predefined actions once certain events are detected. *E-brokerage* [13] and *Amit* [1] are two widely known projects. They detect events and assess situations under timing constraints. *E-brokerage* references an event instance model and *Amit* an event type model. In software engineering, Nentwich et al. propose a framework for repairing inconsistent *XML* documents based on the *xlinkit* tech-

nology [14]. The framework generates interactive repairing options from *FOL*-based constraints [15], but ineffectively supports dynamic computing environments. Capra et al. propose *CARISMA* [5] as a reflective middleware to support mobile applications. It aims to discover policy conflicts, which resembles our work, but assumes accurate contexts available from sensors, which differs from our assumption. Park et al. in [17] address a similar problem and focus on resolving inter-application policy conflicts.

Incremental validation on documents or maintenance on software artifacts has received much attention. Efficient incremental validation techniques for *XML* documents have been studied in [2] and [3] with respect to *DTD*s. The ways to maintain the consistency among software artifacts during development are discussed in [14] and [19]. These works focus mainly on the satisfiability judgment of predefined constraints. The *xlinkit* framework [14] further provides links to help locate inconsistency sources. However, *xlinkit*'s incremental granularity is limited as we discussed earlier.

Our previous work on context inconsistency detection in [24] is based on the Event-Condition-Action (*ECA*) triggering technology, which is suitable to describe simple inconsistent scenarios but weak in expressing complex consistency constraints. In this paper, we propose a more flexible approach, which offers a higher expressive power by taking advantage of *FOL*. Another advantage is the significant performance gain derived from our novel incremental consistency checking semantics.

3. PRELIMINARY

Our context consistency checking allows any data structure for context descriptions. This is because the checking focuses on if consistency constraints are satisfied or not, which does not rely on the underlying context structure. For simplicity, we define context as *ctxt* = (*category*, *fact*, *restriction*, *timestamp*), which only contains the fields we are interested in and will use in later examples:

- *Category* describes the type of the context (e.g., location, temperature or movement).
- *Fact* = (*subject*, *predicate*, *object*) gives the content of the context, where *subject* and *object* are related by *predicate* (simple English sentence structure), e.g., Peter (*subject*) enters (*predicate*) the operating theatre (*object*).
- *Restriction* = (*lifespan*, *site*) specifies the temporal and spatial properties of the context. *Lifespan* represents the time or period the context remains effective. *Site* is the place to which the context relates.
- *Timestamp* records the generation time of the context.

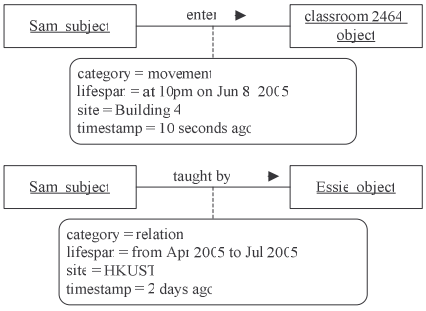

Figure 1. Two example context instances

A *context instance* is defined by instantiating all fields of *ctxt*, while a *context pattern* (or *pattern* for short) is defined by instantiating some of its fields. Each uninstantiated field is set to *any*, which is a special predefined value, meaning that any value is allowed here. Intuitively, each pattern categorizes a set of context instances. The relationship between context instances and patterns is called the *matching* relation, which is mathematically represented by the *belong-to* set operator ∈. Figure 1 illustrates two context instances in *UML* object diagrams, which represent: (1) Sam enters Classroom 2464; (2) Sam is taught by Essie. Figure 2 illustrates two patterns: (1) somebody enters Classroom 2464; (2) somebody is staying in some place. The first context instance in Figure 1 has a matching relation with the first pattern in Figure 2.

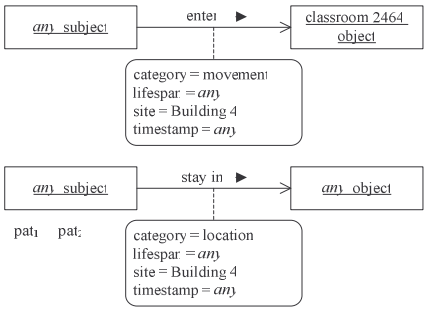

Figure 2. Two example context patterns

4. CONTEXT CONSISTENCY RULES

Consistency constraints on contexts can be generic (e.g., "nobody could be in two different rooms at the same time") or application-specific (e.g., "any goods in the warehouse should have a check-in record before its check-out record"). In our consistency checking, each constraint is expressed by an *FOL* formula as given in Figure 3, where *bfunc* refers to any function that returns true (⊤) or false (⊥). Each expressed constraint is called a *context consistency rule* (or *rule* for short). Note that we are only interested in well-formed rules that contain no free variables.

formula ::= ∀*var*∈*pat* (*formula*) | ∃*var*∈*pat* (*formula*) |
(*formula*) and (*formula*) | (*formula*) or (*formula*) |
(*formula*) implies (*formula*) | not (*formula*) |
bfunc(var_1, …, var_n)

Figure 3. Rule syntax

The rule syntax follows traditional interpretations. For example, ∀*var*∈*pat* (*formula*) represents the constraint that any context instance matched by pattern *pat* must satisfy *formula*. The *formula* definition is recursive until *bfunc* terminals. Thanks to the expressive power of *FOL*, expressing complex constraints becomes easier than using *ECA* counterparts. The constraint in the first example above can be expressed as follows, noting that both pat_1 and pat_2 refer to the second pattern in Figure 2, which is used to retrieve location context instances:

∀γ_1∈pat_1 (not (∃γ_2∈pat_2 (((samesubject(γ_1, γ_2)) and
(samepredicate(γ_1, γ_2, "stay in"))) and
((not (sameobject(γ_1, γ_2))) and (overlaplifespan(γ_1, γ_2))))))

Suppose that we have two context instances recording Sam's location information as illustrated in Figure 4. According to the above rule, they are inconsistent with each other. Our consistency checking returns the result in terms of links. For example, the link {(inconsistent, {ctx_1, ctx_2})} indicates that ctx_1 and ctx_2 cause an inconsistency, in which ctx_1 and ctx_2 are two context instances

matched by pat_1 and pat_2, respectively. In each link, the list of context instances (e.g., {ctx_1, ctx_2}) represents an assignment to the variables (e.g., $\gamma_1 = ctx_1$ and $\gamma_2 = ctx_2$) in the rule, which causes the inconsistency. We assume that each occurrence of a pattern is uniquely identifiable, and every element in such an aforementioned list is annotated with the occurrence of the pattern. This kind of information helps locate problematic context instances. Our goal is to generate such links automatically. The formal definition of links is given in Section 5.2.

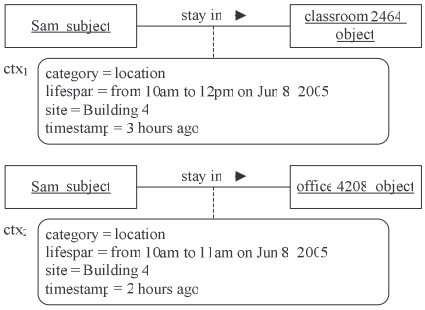

Figure 4. Two inconsistent context instances

5. CONSISTENCY CHECKING

As explained earlier, each context consistency rule is an *FOL* formula extended with context matching. A rule is violated when its associated formula is evaluated to be false. Inconsistency is said to occur if any of the given rules are violated. We give details about how to incrementally evaluate a rule's Boolean value and generate its corresponding links in Subsections 5.1 and 5.2, respectively. To facilitate incremental checking, a stateful context pattern mechanism is proposed to dynamically maintain a pool of interesting context instances, which is explained in Subsection 5.3. To ease our discussion, in the sequel, *rule-based incremental checking* is referred to as *full checking* since it requires the entire formula of every affected rule to be rechecked upon context changes, and *formula-based incremental checking* as *incremental checking* for contrast. The terms checking and rechecking are used interchangeably in the following discussions.

5.1 Boolean Value Evaluation

Consider the following example rule using patterns pat_1, pat_2 and pat_3, and functions f_1 and f_2 (this example is used again in our consistency checking algorithm in Section 6, and its practical meaning can be found in our technical report [25]): $(\forall \gamma_1 \in pat_1 (\exists \gamma_2 \in pat_2 (f_1(\gamma_1, \gamma_2))))$ and (not $(\exists \gamma_3 \in pat_3 (f_2(\gamma_3))))$.

Let V be the set of variables defined in the rules of interest to applications (e.g., $\{\gamma_1, \gamma_2, \gamma_3\}$), and I be the set of context instances. We define a variable assignment: $A = \wp(V \times I)$, which contains mappings between variables and context instances. We also introduce a bind function: $(V \times I) \times A \rightarrow A$, which adds a tuple m formed by a free variable and a context instance to a variable assignment α, i.e., $bind(m,\alpha) = \{m\} \cup \alpha$. Note that bind is a partial function since each variable in α should be unique. Let P be the set of patterns defined in the rules (e.g., $\{pat_1, pat_2\}$). We define the matching function: $\mathcal{M} = P \rightarrow \wp(I)$ such that $\mathcal{M}[pat]$ returns all context instances that match a given pattern *pat*.

Figure 5 gives the Boolean value semantics for the full checking for all formula types (boundary cases are handled when $\mathcal{M}[pat]$ is empty), where F is the set of all formulae. Function $\mathcal{B}[formula]_\alpha$ returns *formula*'s Boolean value by evaluating it under variable assignment α. Given a rule, its initial variable assignment is an empty set $\varnothing$, meaning that no variable has been bound to any value, i.e., a context instance. Function bind may change the variable assignment during the evaluation of the rule's sub-formulae.

$\mathcal{B} : F \times A \rightarrow \{\top, \bot\}$

$\mathcal{B}[\forall var \in pat\ (formula)]_\alpha = \top \wedge \mathcal{B}[formula]_{bind((var, x_1), \alpha)} \wedge \ldots \wedge \mathcal{B}[formula]_{bind((var, x_n), \alpha)} \mid x_i \in \mathcal{M}[pat]$

$\mathcal{B}[\exists var \in pat\ (formula)]_\alpha = \bot \vee \mathcal{B}[formula]_{bind((var, x_1), \alpha)} \vee \ldots \vee \mathcal{B}[formula]_{bind((var, x_n), \alpha)} \mid x_i \in \mathcal{M}[pat]$

$\mathcal{B}[(formula_1)\ \text{and}\ (formula_2)]_\alpha = \mathcal{B}[formula_1]_\alpha \wedge \mathcal{B}[formula_2]_\alpha$

$\mathcal{B}[(formula_1)\ \text{or}\ (formula_2)]_\alpha = \mathcal{B}[formula_1]_\alpha \vee \mathcal{B}[formula_2]_\alpha$

$\mathcal{B}[(formula_1)\ \text{implies}\ (formula_2)]_\alpha = \mathcal{B}[formula_1]_\alpha \rightarrow \mathcal{B}[formula_2]_\alpha$

$\mathcal{B}[\text{not}\ (formula)]_\alpha = \neg \mathcal{B}[formula]_\alpha$

$\mathcal{B}[bfunc(var_1, \ldots, var_n)]_\alpha = bfunc(var_1, \ldots, var_n)_\alpha$

Figure 5. Boolean value semantics for full checking

Existing checking techniques generally reevaluate the entire rule formula whenever a context change is detected. These are inappropriate for dynamic environments because the overhead incurred by repetitive reevaluation of those change-irrelevant subformulae will make the context inconsistency detection less responsive to fast evolving contexts, defying its original purpose. We resolve to evaluate efficiently the Boolean values of rule formulae incrementally.

We assume that consistency rules do not change during incremental formula evaluations. Two kinds of context changes are: (1) *Context addition* occurs when a new context instance is identified by the context management system; (2) *Context deletion* occurs when an old context instance expires due to its freshness requirement, which is a common issue in pervasive computing [24].

We observe that if context changes do not affect a matching result $\mathcal{M}[pat]$, Boolean values of its corresponding formulae also remain unchanged. Therefore, one can focus on those context changes that also lead to the changes of $\mathcal{M}[pat]$.

For illustration, the changes of $\mathcal{M}[pat]$ are examined step by step. There are two types of changes:

- **Add a context instance into** $\mathcal{M}[pat]$**:** A new context instance is identified by the context management system and it belongs to what pattern *pat* is interested in (i.e., they are matched).
- **Delete a context instance from** $\mathcal{M}[pat]$**:** A context instance that previously belonged to $\mathcal{M}[pat]$ expires due to pattern *pat*'s freshness requirement (e.g., 10 seconds in timestamp).

The set of matched context instances for $\mathcal{M}[pat]$ is the accumulated result of additions and deletions of context instances. We incrementally evaluate the rules that need rechecking due to context additions or deletions. We consider in the evaluation only one change made to $\mathcal{M}[pat]$ each time. In the case where multiple changes have occurred, the following strategy is adopted: changes that occur one after another are processed according to their temporal order; changes that occur simultaneously are processed one by one in an arbitrary order. Simultaneous changes occur when multiple context sources generate new context instances at the same time and they are all matched by some patterns. The correctness of this strategy is proved in our technical report [25].

For a given formula, we use the affected function to decide if it is affected by a certain change of $\mathcal{M}[pat]$ and needs reevaluation ($\top$), or not ($\bot$). If this formula or any of its sub-formulae contains pattern *pat*, the function returns true ($\top$); otherwise false ($\bot$). Due to space limitation, we explain in this paper only four formula types using the universal quantifier, and operator, not operator and *bfunc* terminals. A full treatment is in [25].

Let us start our explanation with universal quantifier formulae. As mentioned, we consider in each reevaluation the change made by a single context addition or deletion. The change must affect: (1) the universal quantifier formula itself, or (2) some sub-formula of this formula, or (3) none of them. In the case where the pattern of the universal quantifier formula and those of its sub-formulae are simultaneously affected by a change, we can consider their impacts in turn because the order of their consideration is immaterial [25]. Based on this observation, we identify four cases as illustrated in Figure 6 (note boundary cases in 3 and 4). In incremental checking, we can take advantage of the last evaluation results. Note that $\mathcal{B}_0[formula]_\alpha$ and $\mathcal{M}_0[pat]$ represent the previous values in the last evaluation of $\mathcal{B}[formula]_\alpha$ and $\mathcal{M}[pat]$, respectively.

$\mathcal{B}[\forall var \in pat\ (formula)]_\alpha =$
(1) $\mathcal{B}_0[\forall var \in pat\ (formula)]_\alpha$,
if $\mathcal{M}[pat]$ has no change and affected($formula$) = $\bot$;
(2) $\mathcal{B}_0[\forall var \in pat\ (formula)]_\alpha \wedge \mathcal{B}[formula]_{\text{bind}((var,\ x),\ \alpha)}$
$| \{x\} = \mathcal{M}[pat] - \mathcal{M}_0[pat]$,
if $\mathcal{M}[pat]$ has a context addition change;
(3) $\top \wedge \mathcal{B}_0[formula]_{\text{bind}((var,\ x_1),\ \alpha)} \wedge \ldots \wedge$
$\mathcal{B}_0[formula]_{\text{bind}((var,\ x_n),\ \alpha)} | x_i \in \mathcal{M}[pat]$,
if $\mathcal{M}[pat]$ has a context deletion change;
(4) $\top \wedge \mathcal{B}[formula]_{\text{bind}((var,\ x_1),\ \alpha)} \wedge \ldots \wedge$
$\mathcal{B}[formula]_{\text{bind}((var,\ x_n),\ \alpha)} | x_i \in \mathcal{M}[pat]$,
if affected($formula$) = $\top$

Figure 6. Boolean value semantics for the incremental checking of universal quantifier formulae

$\mathcal{B}[(formula_1)\ \text{and}\ (formula_2)]_\alpha =$
(1) $\mathcal{B}_0[(formula_1)\ \text{and}\ (formula_2)]_\alpha$,
if affected($formula_1$) = affected($formula_2$) = $\bot$;
(2) $\mathcal{B}[formula_1]_\alpha \wedge \mathcal{B}_0[formula_2]_\alpha$, if affected($formula_1$) = $\top$;
(3) $\mathcal{B}_0[formula_1]_\alpha \wedge \mathcal{B}[formula_2]_\alpha$, if affected($formula_2$) = $\top$

Figure 7. Boolean value semantics for the incremental checking of and formulae

$\mathcal{B}[\text{not}\ (formula)]_\alpha =$
(1) $\mathcal{B}_0[\text{not}\ (formula)]_\alpha$, if affected($formula$) = $\bot$;
(2) $\neg\mathcal{B}[formula]_\alpha$, if affected($formula$) = $\top$

$\mathcal{B}[bfunc(var_1, \ldots, var_n)]_\alpha = \mathcal{B}_0[bfunc(var_1, \ldots, var_n)]_\alpha$

Figure 8. Boolean value semantics for the incremental checking of not and *bfunc* formulae

For existential quantifier formulae, the evaluation process is similar. Next, let us consider and formulae. Since there is no pattern in and formulae, the affected function becomes the only factor deciding the evaluation process. Three cases are identified for incrementally evaluating and formulae because this kind of formulae has two sub-formulae and at most one of them can be affected by one given change (although it is possible that both sub-formulae contain a pattern and the two patterns are both affected by the change, we choose to consider the impact to them in turn, the correctness of which has been also proved in [25]). The evaluation process is illustrated in Figure 7. For or and implies formulae, we omit their details here because of the similarity.

Finally, we consider not and *bfunc* formulae (see Figure 8). As a note, the incremental evaluation of Boolean values is relatively intuitive, but the incremental generation of links is challenging.

5.2 Link Generation

Let C = {consistent, inconsistent}. We define *links* [14] as $L = C \times \wp(I)$. Links are hyperlinks connecting multiple consistent or inconsistent elements. They are generated to help locate problematic context instances that cause the inconsistency. A set of context instances that satisfy some consistency rule are connected by a *consistent link* and those that violate some rule are connected by an *inconsistent link*. Consistent and inconsistent links are represented by (consistent, *instset*) and (inconsistent, *instset*), respectively, where *instset* is a set of context instances connected by the link. Links can be manipulated by the following functions:

first((*status*, *instset*)) = *status*
second((*status*, *instset*)) = *instset*
flip : $L \rightarrow L$, where
flip((consistent, *instset*)) = (inconsistent, *instset*)
flip((inconsistent, *instset*)) = (consistent, *instset*)
linkcartesian : $L \times L \rightarrow L$, where
linkcartesian(l_1, l_2) = (first(l_1), second(l_1) $\cup$ second(l_2))
$\otimes : L \times \wp(L) \rightarrow \wp(L)$, where
$l \otimes S = \{\text{linkcartesian}(l, l_1) | l_1 \in S\}$, if $S \neq \varnothing$; otherwise, $\{l\}$
$\underline{\otimes} : \wp(L) \times \wp(L) \rightarrow \wp(L)$, where
$S_1 \underline{\otimes} S_2 = \{\text{linkcartesian}(l_1, l_2) | l_1 \in S_1 \wedge l_2 \in S_2\}$, if $S_1 \neq \varnothing$ and $S_2 \neq \varnothing$; otherwise, $S_1 \cup S_2$

Figure 9. Auxiliary functions for link generation semantics

The flip function flips the status of a link to its opposite. The linkcartesian function takes two links, l_1 and l_2, and produces a new link with the status of l_1 and the union of two context instance sets from l_1 and l_2. We keep the status of l_1 and ignore that of l_2 because l_1 and l_2 should have the same status when we apply this function to them. This assumption holds under the following incremental checking semantics. The $\otimes$ operator takes a link l and a set of links S and produces a new set of links by applying linkcartesian to the pairs formed by link l and every element link in the set S. $\underline{\otimes}$ works similarly but takes two link sets.

We define $\mathcal{L} = F \times A \rightarrow \wp(L)$. $\mathcal{L}[formula]_\alpha$ returns all links of the *formula* of a rule by checking it under some variable assignment α. For a rule to be checked, its initial variable assignment is an empty set $\varnothing$. Like Boolean value evaluation, the bind function may change the variable assignment during the checking of the rule's sub-formulae. Due to space limitation, we only discuss universal, and, not and *bfunc* formulae. The remaining three (existential, or and implies formulae) are similar.

Figure 10 gives the link generation semantics for the full checking of universal and existential quantifier formulae. Note that we focus only on generating interesting links that indicate what causes the inconsistency, i.e., making the given formula evaluated to be false. Due to the negative influence of not operator, we also need

to pay attention to those links that cause the given formula evaluated to be true because the not operator can flip Boolean values (from true to false or from false to true). Based on this observation, we adopt the following strategy. For universal quantifier formulae, we record those links that cause formulae evaluated to be false as inconsistent. For existential quantifier formulae, we record those links that cause formulae evaluated to be true as consistent. Thus, the result includes both inconsistent and consistent links. We may ignore consistent links if we are only interested in inconsistency detection. In the case where universal quantifier formulae are satisfied or existential quantifier formulae are violated, there are no interesting links to be reported [14].

$$\mathcal{L}[\forall var \in pat\ (formula)]_\alpha =$$
$$\{(\text{inconsistent}, \{x\}) \otimes \mathcal{L}[formula]_{\text{bind}((var, x), \alpha)} \mid x \in \mathcal{M}[pat] \wedge \mathcal{B}[formula]_{\text{bind}((var, x), \alpha)} = \bot\}$$
$$\mathcal{L}[\exists var \in pat\ (formula)]_\alpha =$$
$$\{(\text{consistent}, \{x\}) \otimes \mathcal{L}[formula]_{\text{bind}((var, x), \alpha)} \mid x \in \mathcal{M}[pat] \wedge \mathcal{B}[formula]_{\text{bind}((var, x), \alpha)} = \top\}$$

Figure 10. Link generation semantics for the full checking of universal and existential quantifier formulae

The semantics in Figure 10 is only for full checking. Let us continue to discuss how to generate these links incrementally. Our discussion assumes the invariance of the variable assignment α. This is because each affected formula must be fully rechecked if variable assignment α is altered, and incremental checking may not provide much performance gain.

Figure 11 gives the link generation semantics for the incremental checking of universal quantifier formulae. We identify the same four cases as in Figure 6. Note that $\mathcal{L}_0[formula]_\alpha$ represents the previous value in the last checking of $\mathcal{L}[formula]_\alpha$. Similarly, we make use of last checking results. The link generation process for existential quantifier formulae is similar and omitted here.

$\mathcal{L}[\forall var \in pat\ (formula)]_\alpha =$

(1) $\mathcal{L}_0[\forall var \in pat\ (formula)]_\alpha$,
if $\mathcal{M}[pat]$ has no change and affected($formula$) = $\bot$;

(2) $\mathcal{L}_0[\forall var \in pat\ (formula)]_\alpha \cup$
$\{(\text{inconsistent}, \{x\}) \otimes \mathcal{L}[formula]_{\text{bind}((var, x), \alpha)} \mid \{x\} = \mathcal{M}[pat] - \mathcal{M}_0[pat] \wedge \mathcal{B}[formula]_{\text{bind}((var, x), \alpha)} = \bot\}$,
if $\mathcal{M}[pat]$ has a context addition change;

(3) $\{(\text{inconsistent}, \{x\}) \otimes \mathcal{L}_0[formula]_{\text{bind}((var, x), \alpha)} \mid x \in \mathcal{M}[pat] \wedge \mathcal{B}[formula]_{\text{bind}((var, x), \alpha)} = \bot\}$,
if $\mathcal{M}[pat]$ has a context deletion change;

(4) $\{(\text{inconsistent}, \{x\}) \otimes \mathcal{L}[formula]_{\text{bind}((var, x), \alpha)} \mid x \in \mathcal{M}[pat] \wedge \mathcal{B}[formula]_{\text{bind}((var, x), \alpha)} = \bot\}$,
if affected($formula$) = $\top$

Figure 11. Link generation semantics for the incremental checking of universal quantifier formulae

For and, or and implies formulae, generating links is more complicated. We take and formulae for example to explain our ideas. The other two work similarly. One may discard irrelevant information and link up only those context instances that have directly contributed to the Boolean value of a given formula. For example, in formula a∧b, if a is true and b is false, then the formula fails due to b and only due to b, and hence b would be included in the link [16]. Such semantics is a contribution of *xlinkit* and reasonable for and, or and implies formulae. Unfortunately, the integration of such an optimization design with the checking semantics is unclear in *xlinkit* [14].

We deploy the above optimization design in our link generation approach. Our approach tightly integrates the link generation and the optimization to ensure links to exclude non-core information. Then we extend it to support incremental link generation.

Figure 12 gives the link generation semantics for the full checking of and formulae, in which we identify four cases according to the following three principles:

- If both sub-formulae are evaluated to be true, they together contribute to the formula's Boolean value (true);
- If both sub-formulae are evaluated to be false, any of them can decide the formula's Boolean value (false);
- If one sub-formula is evaluated to be true and the other false, then the latter fully decides the formula's Boolean value (false).

$\mathcal{L}[(formula_1)\ \text{and}\ (formula_2)]_\alpha =$

(1) $\mathcal{L}[formula_1]_\alpha \underline{\otimes} \mathcal{L}[formula_2]_\alpha$,
if $\mathcal{B}[formula_1]_\alpha = \mathcal{B}[formula_2]_\alpha = \top$;

(2) $\mathcal{L}[formula_1]_\alpha \cup \mathcal{L}[formula_2]_\alpha$,
if $\mathcal{B}[formula_1]_\alpha = \mathcal{B}[formula_2]_\alpha = \bot$;

(3) $\mathcal{L}[formula_2]_\alpha$, if $\mathcal{B}[formula_1]_\alpha = \top \wedge \mathcal{B}[formula_2]_\alpha = \bot$;

(4) $\mathcal{L}[formula_1]_\alpha$, if $\mathcal{B}[formula_1]_\alpha = \bot \wedge \mathcal{B}[formula_2]_\alpha = \top$

Figure 12. Link generation semantics for the full checking of and formulae

$\mathcal{L}[(formula_1)\ \text{and}\ (formula_2)]_\alpha =$

(1) $\mathcal{L}_0[(formula_1)\ \text{and}\ (formula_2)]_\alpha$,
if affected($formula_1$) = affected($formula_2$) = $\bot$;

(2) a. $\mathcal{L}[formula_1]_\alpha \underline{\otimes} \mathcal{L}_0[formula_2]_\alpha$,
if $\mathcal{B}[formula_1]_\alpha = \mathcal{B}[formula_2]_\alpha = \top$;

b. $\mathcal{L}[formula_1]_\alpha \cup \mathcal{L}_0[formula_2]_\alpha$,
if $\mathcal{B}[formula_1]_\alpha = \mathcal{B}[formula_2]_\alpha = \bot$;

c. $\mathcal{L}_0[formula_2]_\alpha$,
if $\mathcal{B}[formula_1]_\alpha = \top \wedge \mathcal{B}[formula_2]_\alpha = \bot$;

d. $\mathcal{L}[formula_1]_\alpha$,
if $\mathcal{B}[formula_1]_\alpha = \bot \wedge \mathcal{B}[formula_2]_\alpha = \top$,
if affected($formula_1$) = $\top$;

(3) a. $\mathcal{L}_0[formula_1]_\alpha \underline{\otimes} \mathcal{L}[formula_2]_\alpha$,
if $\mathcal{B}[formula_1]_\alpha = \mathcal{B}[formula_2]_\alpha = \top$;

b. $\mathcal{L}_0[formula_1]_\alpha \cup \mathcal{L}[formula_2]_\alpha$,
if $\mathcal{B}[formula_1]_\alpha = \mathcal{B}[formula_2]_\alpha = \bot$;

c. $\mathcal{L}[formula_2]_\alpha$,
if $\mathcal{B}[formula_1]_\alpha = \top \wedge \mathcal{B}[formula_2]_\alpha = \bot$;

d. $\mathcal{L}_0[formula_1]_\alpha$,
if $\mathcal{B}[formula_1]_\alpha = \bot \wedge \mathcal{B}[formula_2]_\alpha = \top$,
if affected($formula_2$) = $\top$

Figure 13. Link generation semantics for the incremental checking of and formulae

To study incremental link generation, three cases are identified based on affected function results and two of them are further classified into four sub-cases based on the above principles. Figure 13 gives a complete description. or and implies formulae are similar and their details are omitted. The remaining two formulae types are not and *bfunc* formulae. We consider them in

Figure 14 and Figure 15. They are relatively simple (note that the not operator flips any consistency status).

$\mathcal{L}[\text{not } (formula)]_\alpha = \{\text{flip}(l) \mid l \in \mathcal{L}[formula]_\alpha\}$

$\mathcal{L}[bfunc(var_1, \ldots, var_n)]_\alpha = \{\}$

Figure 14. Link generation semantics for the full checking of not and *bfunc* formulae

$\mathcal{L}[\text{not } (formula)]_\alpha =$
(1) $\mathcal{L}_0[\text{not } (formula)]_\alpha$, if affected($formula$) = $\perp$;
(2) $\{\text{flip}(l) \mid l \in \mathcal{L}[formula]_\alpha\}$, if affected($formula$) = $\top$

$\mathcal{L}[bfunc(var_1, \ldots, var_n)]_\alpha = \mathcal{L}_0[bfunc(var_1, \ldots, var_n)]_\alpha$

Figure 15. Link generation semantics for the incremental checking of not and *bfunc* formulae

Thus, we discuss all Boolean value and link generation semantics for both full and incremental checking. The remaining problem is how to apply the semantics in a pervasive computing environment. This is discussed in Section 6.

5.3 Stateful Context Patterns

For effective incremental checking, interesting context instances must be retrieved without having to repetitively rescan the entire dataset, i.e., context history. We propose stateful context patterns. Context pattern is a similar concept to *xpath* selector, but they work differently. Given an *xpath* selector, we have to parse the entire *XML* document to find matched content in terms of *DOM* tree nodes. However, the parse is stateless in the sense that *xpath* discards the parsing result to destroy its potential reuse. The entire document must be parsed again next time for matched content even if the changes are minor. In contrast, each pattern, say *pat*, maintains a matching queue to store the last matched context instances (i.e., $\mathcal{M}[pat]$). Whenever a context change is detected, the change is immediately examined to decide its influence on $\mathcal{M}[pat]$. Thus, our approach avoids rescanning the context history in each checking because interesting context instances have already been put in their corresponding matching queues. We observe that *xlinkit* cannot realize such a stateful mechanism for the reason that *xpath* expressions are not independent of each other due to variable references, and hence the last parsing result for each *xpath* expression is difficult to maintain in time, if feasible. In fact, simple constraints, such as uniqueness, represented by *xlinkit* in a meta-model of *XML* documents are generally undecidable [7].

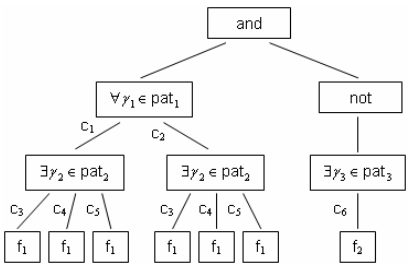

Figure 16. *CCT* of the example rule

In implementation, newly detected context instances are matched against given patterns as explained earlier. Each pattern belongs to one formula or more. Thus, the rules and sub-formulae affected can be determined. Such information is used by our consistency checking algorithm to decide which part of the rule needs rechecking. On the other hand, if a context instance expires with respect to some pattern's freshness requirement, the rules and sub-formulae affected can be also determined similarly and then rechecked using the algorithm explained in the next section.

6. CHECKING ALGORITHM

6.1 Consistency Computation Tree

To facilitate the checking of *FOL* rules, we convert each rule into a *consistency computation tree* (*CCT*). All functions are converted into *leaf nodes*. Other operators like ∀, ∃, and, or, implies and not are converted into *internal nodes*. Each internal node may have one or multiple branches. Uni-operators (e.g., not) only have one branch; bi-operators (e.g., and, or and implies) have two branches; for multi-operators (e.g., ∀ and ∃), their branch numbers are determined by the size of the individual $\mathcal{M}[pat]$ results.

The conversion is straightforward. Suppose that we have a rule $(\forall \gamma_1 \in \text{pat}_1\ (\exists \gamma_2 \in \text{pat}_2\ (\text{f}_1(\gamma_1, \gamma_2))))$ and (not $(\exists \gamma_3 \in \text{pat}_3\ (\text{f}_2(\gamma_3))))$, and $\mathcal{M}[\text{pat}_1] = \{\text{c}_1, \text{c}_2\}$, $\mathcal{M}[\text{pat}_2] = \{\text{c}_3, \text{c}_4, \text{c}_5\}$ and $\mathcal{M}[\text{pat}_3] = \{\text{c}_6\}$ (note that $\text{c}_1, \text{c}_2, \ldots, \text{c}_6$ represent context instances). Then its corresponding *CCT* is illustrated in Figure 16.

For a ∀ or ∃ node, each of its branches represents the sub-formula of the formula the node represents with a certain variable binding (e.g., the leftmost branch of node $\forall \gamma_1 \in \text{pat}_1$ corresponds to a variable binding of $\gamma_1 = \text{c}_1$). The initial variable assignment for the root node is an empty set ∅. When we traverse the *CCT* from the top down, each node adds certain variable bindings from its branches (if any) to the variable assignment.

6.2 Full Consistency Checking

The full consistency checking algorithm consists of three steps:
- Create the *CCT* of the given rule that needs checking;
- Calculate the Boolean value of the rule (post-order traversal);
- Calculate the links generated (post-order traversal).

CCT creation and traversal require processing the entire tree.

6.3 Incremental Consistency Checking

CCT creation and traversal in the incremental checking algorithm only require processing small parts of the tree. We term a *critical node* as a node that contains an occurrence of a pattern affected by the context change. Although we consider one $\mathcal{M}[pat]$ change only each time, there may be several critical nodes in the tree. We observe that all critical nodes of the same occurrence of a pattern must be at the same level. This is because they are all derived from the same sub-formula that contains the affected pattern but with different variable bindings. The checking algorithm works in three steps: (i) adjust the *CCT* of the given rule, (ii) adjust the Boolean value of the rule, and (iii) adjust the links generated.

6.3.1 Adjustment of CCT

Each critical node must be a universal or an existential quantifier node, and the adjustment would change its branches. Let *N* be a critical node and *pat* be its pattern's occurrence at *N*. There are two kinds of adjustment to the *CCT* when $\mathcal{M}[pat]$ changes:
- **Addition of a context instance *x* to $\mathcal{M}[pat]$:** Add a new branch corresponding to *x* to *N*.
- **Deletion of a context instance from $\mathcal{M}[pat]$:** Delete an old branch corresponding to *x* from *N*.

Each critical node has to execute such adjustment. For example:

- If $\mathcal{M}[\mathsf{pat}_2]$ changes from $\{\mathsf{c}_3, \mathsf{c}_4, \mathsf{c}_5\}$ to $\{\mathsf{c}_3, \mathsf{c}_4, \mathsf{c}_5, \mathsf{c}_7\}$, the result *CCT* is illustrated in Figure 17.
- If $\mathcal{M}[\mathsf{pat}_1]$ changes from $\{\mathsf{c}_1, \mathsf{c}_2\}$ to $\{\mathsf{c}_2\}$, the result *CCT* is illustrated in Figure 18.

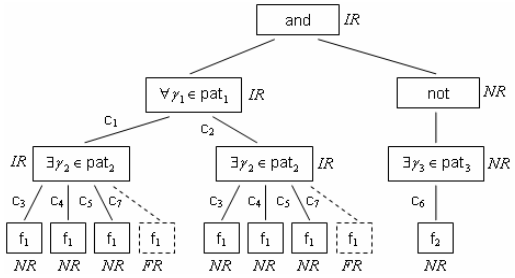

Figure 17. *CCT* Adjustment – add branches (dashed branches are newly added)

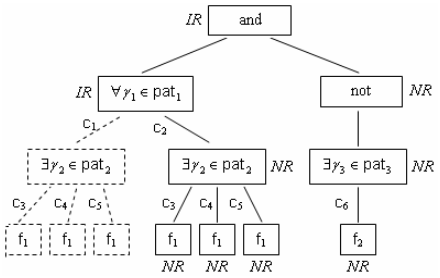

Figure 18. *CCT* Adjustment – delete branches (the dashed branch is newly deleted)

6.3.2 Adjustment of Boolean Values and Links

First, if the *CCT* adjustment is to add branches, all nodes on the added branches (excluding critical nodes) must be fully checked (post-order traversal) for Boolean values and links because their variable assignments contain the new change. Then, whether the *CCT* adjustment is to add or delete branches, all nodes on the paths from critical nodes up to the root node (including critical nodes) can be incrementally rechecked (post-order traversal) for new Boolean values and links because each of them has at least one branch checked/rechecked. There is no need to check/recheck other nodes. Figure 17 and Figure 18 show how to adjust Boolean values and links for the above examples (*FR*: fully checked; *IR*: incrementally rechecked; *NR*: no checking/rechecking required).

6.4 Time Complexity Analysis

Let the node number of a *CCT* be n (including all leaf nodes and internal nodes). It can be inferred that a full post-order traversal of the *CCT* takes $O(n)$ time. Upon context changes, suppose that the node number changes from n to n_1 because of adding or deleting branches. Full checking has to take $O(3n_1) = O(n_1)$ time to recreate the *CCT* and recalculate Boolean values and links.

For incremental checking, computation is spent mainly on *CCT* adjustment. Let the number of nodes on the paths from critical nodes to the root node be n_2. There are two cases:

- **Add branches:** The time cost has two parts: (1) adjust $n_1 - n$ nodes on the added branches (including node creation, Boolean value evaluation and link generation); (2) adjust n_2 nodes on the paths from critical nodes to the root node (including Boolean value reevaluation and link regeneration). Thus, the total time cost is $O(3(n_1 - n) + 2n_2) = O(3n_1 - 3n + 2n_2)$.
- **Delete branches:** The deletion takes $O(1)$ time. Other time is spent on adjusting n_2 nodes on the paths from critical nodes to the root node (including Boolean value reevaluation and link regeneration). Thus the total time cost is $O(1 + 2n_2)) = O(n_2)$.

To further analyze the relationships among n, n_1 and n_2, let the distance from the critical nodes to the root node be h (note that all critical nodes are at the same level) and the number of critical nodes be c. We argue that the following two conditions hold:

- $O(c) \leqslant |n_1 - n| \leqslant O(n)$: When all critical nodes are right above leaf nodes, there are only c new leaf nodes to be added or c old leaf nodes to be deleted. Thus $|n_1 - n| = O(c)$. When all critical nodes are the root node itself, there is a very large branch to be added or deleted, whose size is near $O(n)$. Thus $|n_1 - n| = O(n)$. Other cases are in between.
- $O(h + c) \leqslant n_2 \leqslant O(hc)$: The paths from the critical nodes to the root node eventually merge into one path, the lowest node of which is called the *split node*. When the split node is right above critical nodes (or the split node is the only critical node), most nodes on the paths are shared by all paths. Thus $n_2 = O(h + c)$. When the split node is the root node, most nodes are unshared. Thus $n_2 = O(hc)$. Other cases are in between.

The best case is reached when there is only one critical node ($c = 1$), which is also the split node and right above leaf nodes. The worst case is reached when the split node and the only critical node ($c = 1$) are the root node itself.

The best case (*FC*: full checking, *ICADD*: incremental checking – add branches, *ICDEL*: incremental checking – delete branches):

- *FC*: $O(n_1) = O(n + c) = O(n)$
- *ICADD*: $O(3n_1 - 3n + 2n_2) = O(2h + 5c) = O(h)$
- *ICDEL*: $O(n_2) = O(h + c) = O(h)$

The worst case:

- *FC*: $O(n_1) = O(2n) = O(n)$
- *ICADD*: $O(3n_1 - 3n + 2n_2) = O(3n + 2hc) = O(3n + 2h) = O(n)$
- *ICDEL*: $O(n_2) = O(hc) = O(h)$

From the above analysis, incremental checking is more efficient than full checking because h is much less than n in practice.

7. EXPERIMENTATION

The goal of our experiments is to estimate and compare the performance between the full checking algorithm (or *FC* for short) and the incremental checking algorithm (or *IC* for short). *FC* is actually the rule-based incremental checking we discussed earlier. We used our own implementations. The reason we do not convert context instances into *XML* format to compare our approach with the *xlinkit* approach directly is that the conversion time is significant, compared to the actual checking time, and after conversion we cannot make use of the stateful pattern mechanism.

The experiments were performed on a Pentium IV 1.3-GHz machine running Windows XP Professional SP2. The test environment consisted of the middleware part and client part. A simulated context source generated new context instances for the middleware at a constant rate. To study the impact of different workloads, we gradually increased the time interval of generating context instances from 0.1 seconds to 0.4 seconds at a step of 0.1 seconds. We also collected experimental results when the total number of processed context instances varied from 100 to 500 at a step of 100. Note that the number of context changes doubles that of received context instances since each received context instance is

processed by the middleware twice: once as context addition (or *ADD* for short) and once as context deletion (or *DEL* for short) due to its expiration some time later. We fix the freshness requirement to be five seconds, meaning that each received context instance remains valid for five seconds, during which it can participate in any rule checking. We choose *Cabot* [24] as our infrastructure to provide context services. The consistency checking works as a third-party context-filtering service in the middleware.

We designed nine consistency rules whose *CCT*s have a height of four to six levels (note that the example *CCT* in Section 6 only has three levels). The rules cover all *FOL* formula types. Every rule contains two universal or existential quantifier sub-formulae because they would cause the greatest complexity to consistency checking. The rules include 18 patterns totally. Each context instance was specially designed so that it could match at least one pattern. When a pattern is matched, it activates the checking of relevant rules that are built on it. Among all context instances, 66.7% of them contribute to context inconsistency, i.e., they cause the inconsistent status during checking.

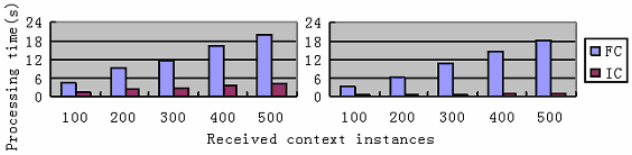

Figure 19. Performance comparison between *FC* and *IC* (left: *ADD*, right: *DEL*, time interval: 200ms)

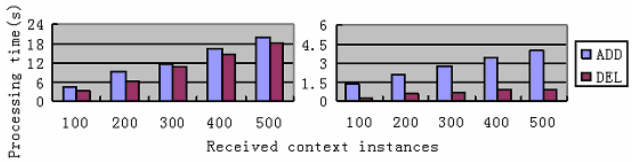

Figure 20. Performance comparison between *ADD* and *DEL* (left: *FC*, right: *IC*, time interval: 200ms)

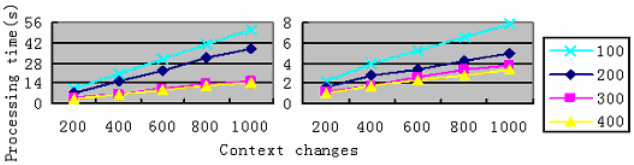

Figure 21. Performance comparison under different workloads (left: *FC*, right: *IC*)

Figure 19 illustrates the performance comparison result between *FC* and *IC* when the time interval is fixed to 0.2 seconds (left: processing the *ADD* scenario, right: processing the *DEL* scenario). Both graphs show that *IC* is superior to *FC*. *IC* is approximately five times faster than *FC* when processing the *ADD* scenario and even faster when processing the *DEL* scenario.

Figure 20 illustrates the performance comparison result between processing *ADD* and *DEL* scenarios when the time interval is fixed to 0.2 seconds (left: *FC*, right: *IC*). Two results are quite different. For *FC* (left), the processing times for *ADD* and *DEL* scenarios are quite close; but for *IC* (right), there is a big difference. It is understandable because from the analysis in the last section *FC*'s time complexity is $O(n)$ for processing both *ADD* and *DEL* scenarios, but *IC*'s time complexity is between $O(h)$ and $O(n)$ for processing the *ADD* scenario and always $O(h)$ for processing the *DEL* scenario. In practice, h is much less than n. Thus *IC* has an even greater advantage over *FC* when processing the *DEL* scenario. The experiment confirms our analysis results.

Figure 21 illustrates the performance comparison result under different workloads by changing the time interval from 0.1 seconds to 0.4 seconds (left: *FC*, right: *IC*). Although two algorithms' performance results differ a lot (about seven times), their trends behave similarly: when the time interval is decreased, the processing time is increased accordingly. This is because we fixed the freshness requirement to be five seconds. If we have more context instances arriving at the middleware during a time unit (i.e., a smaller time interval), there would then be more legitimate context instances for consistency checking (i.e., longer matching queues). Thus, the processing time must be increased since all legitimate context instances have to be examined.

The above experimental results suggest that *IC* is more suitable than *FC* is for dynamic environments where context changes frequently. The results are subject to our own implementation. More experiments on other implementations are needed to conclude if they follow similar trends as we have reported in this paper.

8. CONCLUSION

We have proposed a novel formula-based incremental checking approach for context consistency management for pervasive computing. Our approach carefully distinguishes the core part from the non-core part to improve the checking performance. The core part reevaluates the affected sub-formulae since the incurred re-evaluation is inevitable. The non-core reuses the previously evaluated and still valid evaluation results to substitute the effort of re-computing the results of unaffected sub-formulae.

Our approach is heavily influenced by the *xlinkit* technology. The notation of links is from *xlinkit*, but we use a different approach to generate them. The idea of auxiliary functions is also adapted from *xlinkit*. The difference lies in that their original use in *xlinkit* is only for rule-based incremental checking while we have adapted them to our incremental checking with a finer granularity. *xlinkit* is stateless among consecutive checking of the same formula, for which our tightly integrated strategy is stateful and thus we realize the reuse of stateful information in our approach.

We have also studied the intrinsic imperfectness of pervasive contexts, and identified a mandate of effective context consistency checking. We have proposed a formal semantics for incrementally checking first order logic rules to address the problem of context inconsistency detection. Based on the semantics, we have presented an efficient consistency checking algorithm and evaluated our approach against conventional checking techniques on our *Cabot* middleware. The experiments justified our efforts.

Currently, the link generation process still suffers from a few limitations. It may produce redundant links when a pattern appears more than once in a rule. Simple approaches to eliminating them (e.g., checking link permutations [14]) are computationally expensive. We plan to study the problem by inferring causal relations among inconsistencies. Another limitation is the space used for keeping last checking results. We trade this for the time reduction in subsequent checking. We will consider how to alleviate the state explosion problem in future, which is serious when all matching queues are very long. Currently, the use of rules is limited to express necessary conditions for context consistency,

which can be easily extended in future to express sufficient conditions for context inconsistency. Besides, we plan to apply our incremental checking technology to situation assessment, which is critical to context-aware applications because they need to respond quickly to environmental changes.

9. REFERENCES

[1] Adi, A. and Etzion, O. Amit: The Situation Manager. *VLDB Journal 13(2)*, pp. 177–203, 2004.

[2] Balmin, A., Papakonstantinou, Y., and Vianu, V. Incremental Validation of XML Documents. *ACM Transactions on Database Systems 29(4)*, pp. 710–751, Dec 2004.

[3] Barbosa, D., Mendelzon, A.O., Libkin, L., Mignet, L., and Arenas, M. Efficient Incremental Validation of XML Documents. *Proceedings of the 20th International Conference on Data Engineering*, pp. 671–682, Boston, US, Mar 2004.

[4] Brumitt, B., Meyers, B., Krumm, J., Kern, A., and Shafer, S. EasyLiving: Technologies for Intelligent Environments. *Proceeding of the 2nd International Symposium on Handheld and Ubiquitous Computing*, pp. 12–29, Bristol, England, 2000.

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

[6] Dey, A.K., Abowd, G.D., and Salber, D. A Context-Based Infrastructure for Smart Environments. *Proceedings of the 1st International Workshop on Managing Interactions in Smart Environments*, pp. 114–128, Dublin, Ireland, Dec 1999.

[7] Fan W.F. On XML Integrity Constraints in the Presence of DTDs. *Journal of the ACM 49(3)*, pp. 368–406, May 2002.

[8] Griswold, W.G., Boyer, R., Brown, S.W., and Tan, M.T. A Component Architecture for an Extensible, Highly Integrated Context-Aware Computing Infrastructure. *Proceedings of the 25th International Conference on Software Engineering*, pp. 363–372, Portland, US, May 2003.

[9] Harter, A., Hopper, A., Steggles, P., Ward, A., and Webster, P. The Anatomy of a Context-Aware Application. *Proceedings of the 5th ACM/IEEE International Conference on Mobile Computing and Networking*, pp. 59–68, Seattle, US, Aug 1999.

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

[11] Judd, G. and Steenkiste, P. Providing Contextual Information to Pervasive Computing Applications. *Proceedings of the 1st IEEE International Conference on Pervasive Computing and Communications*, pp. 133–142, Dallas, US, Mar 2003.

[12] Julien, C. and Roman, G.C. Egocentric Context-Aware Programming in Ad Hoc Mobile Environments. *Proceedings of the 10th International Symposium on the Foundations of Software Engineering*, pp. 21–30, Charleston, US, Nov 2002.

[13] Mok, A.K., Konana, P., Liu, G., Lee, C.G., and Woo, H. Specifying Timing Constraints and Composite Events: An Application in the Design of Electronic Brokerages. *IEEE Transactions on Software Engineering 30(12)*, pp. 841–858, Dec 2004.

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

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

[16] Nentwich, C., Emmerich, W., and Finkelstein, A. Flexible Consistency Checking. *ACM Transactions on Software Engineering and Methodology 12(1)*, pp. 28–63, Jan 2003.

[17] Park, I., Lee, D., and Hyun, S.J. A Dynamic Context-Conflict Management Scheme for Group-Aware Ubiquitous Computing Environments, *Proceedings of the 29th International Computer Software and Applications Conference*, pp. 359–364, Edinburgh, UK, Jul 2005.

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

[19] Reiss, S. Incremental Maintenance of Software Artifacts. *Proceedings of the 21st IEEE International Conference on Software Maintenance*, Budapest, Hungary, Sep 2005.

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

[21] Schmidt, A., Aidoo, K.A., Takaluoma, A., Tiomela, U., Van, L.K., and Van, D.V.W. Advanced Interaction in Context. *Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing*, pp. 89–101, Karlsruhe, Germany, Sep 1999.

[22] Sousa, J.P. and Garlan, D. Aura: An Architectural Framework for User Mobility in Ubiquitous Computing Environments. *Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture*, pp. 29–43, Montreal, Canada, Aug 2002.

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

[24] Xu, C. and Cheung, S.C. Inconsistency Detection and Resolution for Context-Aware Middleware Support. *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.

[25] Xu, C. and Cheung, S.C. Incremental Context Consistency Checking. *Technical Report HKUST-CS05-15*. Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, China, Oct 2005.