Incremental Consistency Checking for Pervasive Context[1]

Chang Xu Shing-Chi Cheung Wing-Kwong Chan
Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong, China
{changxu, scc, wkchan}@cse.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 should be aware of incessant context changes in their environments and adjust their behavior accordingly. As such 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 the *Cabot* middleware.

Categories and Subject Descriptors

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

General Terms

Algorithms, Performance, Theory, Verification.

Keywords

Context Management, Incremental Consistency Checking, Pervasive Computing.

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 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, simply because the context mistakenly dictates that the environment is a football match. This poses a natural requirement on *consistent contexts* [24].

A *context* of a computational 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, but it differs from data in conventional databases in that the latter are integral and consistent during a database transaction. On the other hand, it is difficult for contexts to be always 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. A detailed analysis of the reasons on inevitably imperfect contexts for pervasive computing can be found in existing work [24].

As a result, one needs to deal with inconsistent contexts in pervasive computing. To ensure 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 query 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 one example. *Call Forwarding* [23] is a location-aware application, which aims to forward incoming calls to target callees with phones nearby. *Call Forwarding* assumes phone receptionists knowing the callees' current locations. Generally, raw sensory data are collected by a 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") need 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 actions. In this example, the goal of acquiring precise location contexts becomes the basis to reject inconsistent location contexts that violate pre-specified consistency constraints.

The checking of consistency constraints, or *consistency management*, has been recognized as an important research issue by software engineering and programming communities [14]. Generally, consistency constraints are expressed as rules and checked over interesting contents (e.g., documents and programs) such that any violation of rules, called *inconsistency*, can be detected. The detected inconsistencies are 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 attention is paid mainly to interesting content changes and checking is conducted eagerly or lazily to identify whether 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 each rule, *xlinkit* employs *xpath* selectors to retrieve interesting contents from *XML* documents. This approach is intuitive in the sense that each time the entire set of *XML* documents is checked against all rules. We term this

[1] Crafted version: Made in Feb 2017 (originally published at ICSE in May 2006). Contact: Chang Xu (changxu@nju.edu.cn).

type of checking *all-document checking*.

However, all-document checking is computationally expensive when applied to a large dataset that requires frequent checking. *xlinkit*'s improved work [16] can identify those rules whose targeted contents are potentially affected by a given change. Only these affected rules need rechecking 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 contents affects only 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 change. In practice, a rule typically contains universal or existential sub-formulae, in which variable assignments are subject to change [16]. If the granularity of checking can be confined to sub-formula, it would save much rechecking effort. Second, each affected rule has to be rechecked against the entire dataset or documents even for a small change. This is because the last retrieved content from *xpath* selectors was not stored, and thus could not be reused in subsequent checking. The above two limitations present a major challenge when one applies rule-based incremental checking to dynamic pervasive computing environments, in which applications require timely responses to context changes but typically context changes are rapid and each change is marginal.

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

Although various studies [2][3][14][19] have partially addressed the second challenge, the first one 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 Boolean value semantics and link generation semantics for incremental consistency checking;
- The proposal of an efficient and incremental algorithm to detect context inconsistency for pervasive computing.

The remainder of this paper is organized as follows. Section 2 discusses recent related work. Sections 3 and 4 introduce preliminary concepts on context modeling and consistency checking, respectively. Section 5 elaborates on 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. Finally, Section 8 concludes this 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 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 piece of representative work is *Context Toolkit* [6]. It assists developers by providing abstract components to obtain and process context data from sensors. *Context Toolkit* falls short of supporting highly integrated applications, and so Hence Griswold et al. [8] proposed to apply a hybrid mediator-observer pattern in an application's architecture. To facilitate context-aware computing, Henricksen et al. [10] presented a multi-layer framework that supports both conditional selection and invocation of program components. Ranganathan et al. [18] discussed how to resolve potential semantic contradictions in contexts by reasoning based on first-order predicate calculus and Boolean algebra. These pieces of work 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 database and software engineering fields. In the *AI* field, expert systems for supporting strategy formulation and decision-making are common. Much effort has been made on the evidence aggregation problem such that the systems can develop strategies over contradicting rules [24]. In the active database field, advanced event detection techniques have been 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 temporal constraints. *E-brokerage* references an event instance model and *Amit* uses an event type model. In the software engineering field, Nentwich et al. proposed a framework for repairing inconsistent *XML* documents based on the *xlinkit* technology [14]. The framework generates interactive repairing options from *FOL*-based constraints [15], but ineffectively supports dynamic computing environments. Capra et al. proposed *CARISMA* [5] as a reflective middleware to support mobile applications. It aims to discover policy conflicts. This resembles our work, but assumes accurate contexts available from sensors, and this differs from our assumption. Park et al. [17] addressed a similar problem and focused 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 with respect to *DTD*s [2][3]. The ways to maintain the consistency among software artifacts during development have also been discussed [14] [19]. These pieces of work focus mainly on the satisfiability evaluation 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 [24] is based on the Event-Condition-Action (*ECA*) triggering technology, which is suitable to specify simple inconsistent scenarios but could be painful in specifying 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

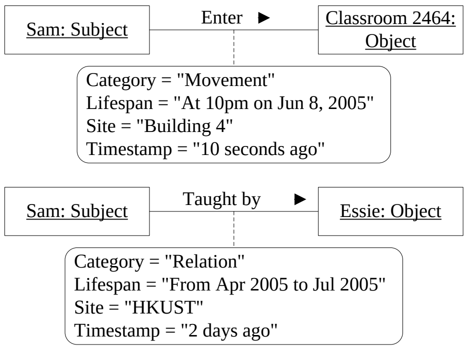

Figure 1. Two example context instances.

$$
\begin{aligned}
formula ::= {}& \forall var \in pat\ (formula) \mid \exists var \in pat\ (formula) \mid \\
& (formula)\ \text{and}\ (formula) \mid (formula)\ \text{or}\ (formula) \mid \\
& (formula)\ \text{implies}\ (formula) \mid \text{not}\ (formula) \mid \\
& bfunc(var_1, \ldots, var_n).
\end{aligned}
$$

Figure 3. Rule syntax.

is the significant performance gain achieved by our novel incremental consistency checking semantics.

3. PRELIMINARY

Our context consistency checking allows any data structure for context specification. This is because the checking focuses on whether 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 contains only fields we are interested in, and use it in subsequent discussions:

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

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 value, meaning “do not care”. Intuitively, each pattern recognizes a set of context instances. The relationship between context instances and patterns is called *matching*, which is mathematically represented by the *belong-to* set operator $\in$. Figure 1 illustrates two context instances in a *UML* object diagram, 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”. It is easy to observe that the first context instance in Figure 1 matches the first pattern in Figure 2.

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 item in the warehouse should have a check-in record before its check-out record”). In our consistency checking,

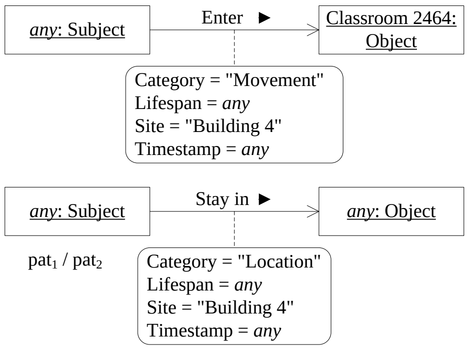

Figure 2. Two example context patterns.

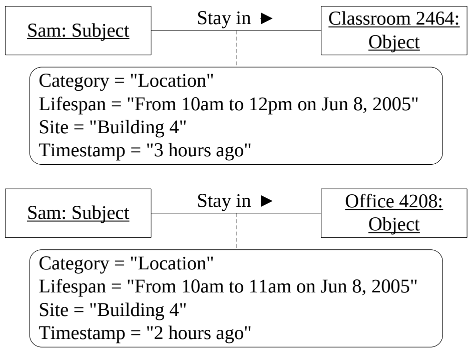

Figure 4. Two inconsistent context instances.

each constraint is expressed by an *FOL* formula as given in Figure 3, where *bfunc* refers to any function that returns true ($\top$) or false ($\bot$). 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 variable.

The rule syntax follows traditional interpretations. For example, $\forall var \in pat$ (*formula*) represents a 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 our earlier *ECA* counterparts [24]. The constraint in the first example 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:

$$
\begin{aligned}
&\forall \gamma_1 \in pat_1\ (\text{not}\ (\exists \gamma_2 \in pat_2\ (((\text{sameSubject}(\gamma_1, \gamma_2))\ \text{and} \\
&\quad (\text{samePredicate}(\gamma_1, \gamma_2, \text{"stay in"})))\ \text{and} \\
&\quad ((\text{not}\ (\text{sameObject}(\gamma_1, \gamma_2)))\ \text{and}\ (\text{overlapLifespan}(\gamma_1, \gamma_2)))))).
\end{aligned}
$$

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, 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 (i.e., {ctx_1, ctx_2}) represents an assignment to the variables (i.e., $\gamma_1 = ctx_1$ and $\gamma_2 = ctx_2$) in the rule, which exactly causes the inconsistency. We assume that each pattern is uniquely identifiable, and each element in such an aforementioned list is annotated with its corresponding pattern. This kind of information helps locate problematic context instances. Our goal is to generate such links

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

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

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

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

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

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

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

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

Figure 5. Boolean value semantics for full checking.

automatically and efficiently. The formal definition of link is given later in Section 5.2.

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 false. Inconsistency is said to occur if any of given rules is violated. We give details about how to incrementally evaluate a rule's Boolean value and generate its corresponding links in Sections 5.1 and 5.2, respectively. To facilitate incremental checking, we propose a stateful context pattern mechanism to dynamically maintain a pool of interesting context instances, which is explained in Section 5.3. To ease our discussion, we refer to *rule-based incremental checking* 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 may be used interchangeably in the following.

5.1 Boolean Value Evaluation

Consider the following example rule with patterns pat_1, pat_2 and pat_3, and functions f_1 and f_2 (the example is also used later in Section 6 and its practical meaning can be found in our technical report [25]):

$(\forall \gamma_1 \in \text{pat}_1\ (\exists \gamma_2 \in \text{pat}_2\ (\text{f}_1(\gamma_1, \gamma_2))))$ and $(\text{not}\ (\exists \gamma_3 \in \text{pat}_3\ (\text{f}_2(\gamma_3))))$.

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 available context instances. We define a variable assignment: $A = \wp(V \times I)$, which contains mappings between variables and context instances. We also introduce the bind function: $(V \times I) \times A \rightarrow A$, which adds a tuple m formed by a free variable and a context instance to an existing variable assignment α to form a new one, i.e., $\text{bind}(m, \alpha) = \{m\} \cup \alpha$. 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., $\{\text{pat}_1, \text{pat}_2\}$). We define the matching function: $\mathcal{M} = P \rightarrow \wp(I)$ such that $\mathcal{M}[pat]$ returns the set of context instances matching a given pattern *pat*.

Figure 5 gives our Boolean value semantics for full checking for all formula types (note that boundary cases have been 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 ϕ, meaning that no variable has been bound to any

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

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

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

(3) $\top \wedge \mathcal{B}_0[formula]_{\text{bind}((var, x_1), \alpha)} \wedge \ldots \wedge \mathcal{B}_0[formula]_{\text{bind}((var, x_n), \alpha)} \mid x_i \in \mathcal{M}[pat]$,
if $\mathcal{M}[pat]$ has a context deletion change;

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

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

value (context instance). Function bind may change the variable assignment during the evaluation of the rule's sub-formulae.

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

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

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

For ease of discussion, changes to $\mathcal{M}[pat]$ are examined step by step. There are two types of such change:

- **Add a context instance into $\mathcal{M}[pat]$:** A new context instance is identified by the system and it matches pattern *pat*.
- **Delete a context instance from $\mathcal{M}[pat]$:** An existing context instance belonging to $\mathcal{M}[pat]$ expires due to pattern *pat*'s freshness need (e.g., 10 seconds as specified in the *timestamp* field).

The set of matched context instances for $\mathcal{M}[pat]$ is thus the accumulated result of addition and deletion of context instances. We incrementally evaluate the rules that need rechecking due to context addition or deletion. We consider in the evaluation only one change made to $\mathcal{M}[pat]$ each time. In the case where multiple changes have occurred, we take the following strategy: changes that occur one after another are processed according to their temporal orders; 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. We have proved the correctness of this strategy in our technical report [25].

For a given formula, we use the affected function to decide whether

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

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

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

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

Figure 7. Boolean value semantics for incremental checking of and formula.

$\text{first}((status, instSet)) = status$.
$\text{second}((status, instSet)) = instSet$.
flip: $L \to L$, where
 $\text{flip}((\text{consistent}, instSet)) = (\text{inconsistent}, instSet)$, and
 $\text{flip}((\text{inconsistent}, instSet)) = (\text{consistent}, instSet)$.
linkCartesian: $L \times L \to L$, where
 $\text{linkCartesian}(l_1, l_2) = (\text{first}(l_1), \text{second}(l_1) \cup \text{second}(l_2))$.
$\otimes$: $L \times \wp(L) \to \wp(L)$, where
 $l \otimes S = \{\text{linkCartesian}(l, l_1) \mid l_1 \in S\}$, if $S \neq \phi$; otherwise, $\{l\}$.
$\underline{\otimes}$: $\wp(L) \times \wp(L) \to \wp(L)$, where
 $S_1 \underline{\otimes} S_2 = \{\text{linkCartesian}(l_1, l_2) \mid l_1 \in S_1 \wedge l_2 \in S_2\}$, if $S_1 \neq \phi$ and $S_2 \neq \phi$; otherwise, $S_1 \cup S_2$.

Figure 9. Auxiliary functions for link generation.

it is affected by a certain change to $\mathcal{M}[pat]$ and thus needs reevaluation ($\top$), or not ($\perp$). If this formula or any of its sub-formulae contains pattern *pat*, the function returns true ($\top$), and otherwise false ($\perp$). Due to space limitation, we explain in this paper only four formula types, i.e., formula with universal quantifier, and operator, not operator and *bfunc* terminal. A full treatment can also be found in our technical report [25].

Let us start our explanation with universal quantifier formula. As mentioned, we consider in each evaluation 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 all affected by a change, we consider their impacts in turn because the order of their consideration is immaterial (proof in [25]). Based on this observation, we identify four cases as given in Figure 6 (note that boundary cases have been handled in (3) and (4)). In incremental checking, one can take advantage of last evaluation results. Note that $\mathcal{B}_0[formula]_\alpha$ and $\mathcal{M}_0[pat]$ represent previous values in the last evaluation of $\mathcal{B}[formula]_\alpha$ and $\mathcal{M}[pat]$, respectively.

For existential quantifier formula, its evaluation process is similar and thus omitted. We next consider and formula. Since there is no pattern in an and formula, the affected function becomes the only factor deciding its evaluation process. We identify three cases for incrementally evaluating an and formula because such formula 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 both patterns are affected by the change, we choose to consider the impact to them in turn, and its correctness has been proved in our technical report [25]). The evaluation process is given in Figure 7. For or and implies formulae, we omit their details due to similarity.

Finally, we consider not and *bfunc* formulae (see Figure 8). Their semantics are straightforward. We note that incremental evaluation of Boolean values for FOL formulae is relatively intuitive, but incremental generation of links is more challenging, as we explain below.

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

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

(2) $\neg\mathcal{B}[formula]_\alpha$, if $\text{affected}(formula) = \top$.

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

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

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

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

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 a consistency rule are connected by a *consistent link* and those that violate a 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 functions given in Figure 9.

The flip function reverses the status of a link to its opposite. The linkCartesian function takes two links, l_1 and l_2, as input and returns a new link with the status of l_1 and union of two context instance sets from l_1 and l_2, respectively. 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 as input and returns a new set of links by applying linkCartesian to the pairs formed by link l and every element link in set S. The $\underline{\otimes}$ operator works similarly but takes two link sets as input.

We define $\mathcal{L} = F \times A \to \wp(L)$. $\mathcal{L}[formula]_\alpha$ returns all links for *formula* by checking it under variable assignment α. For a rule under checking, its initial variable assignment is an empty set ϕ. 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 discuss only universal, and, not and *bfunc* formulae. The remaining three (existential, or and implies formulae) are similar.

Figure 10 gives our link generation semantics for full checking of universal and existential quantifier formulae. Note that we focus only on generating interesting links that explain what has caused the inconsistency, i.e., making a given formula evaluated to false. Due to the negative effect of the not operator, we also need to pay attention to those links that have caused a given formula evaluated

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

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

(2) $\mathcal{L}_0[\forall var \in pat\ (formula)]_{\alpha} \cup$
  $\{\ (\text{inconsistent}, \{x\}) \otimes \mathcal{L}[formula]_{\text{bind}((var,\, x),\, \alpha)}$
  $\mid \{x\} = \mathcal{M}[pat] \setminus \mathcal{M}_0[pat] \wedge \mathcal{B}[formula]_{\text{bind}((var,\, x),\, \alpha)} = \bot\ \}$,
  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 $\text{affected}(formula) = \top$.

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

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

(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 $\mathcal{B}[formula_1]_{\alpha} = \top$ and $\mathcal{B}[formula_2]_{\alpha} = \bot$;

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

Figure 12. Link generation semantics for full checking of and formula.

to true since 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 a universal quantifier formula, we record those links that have caused the formula evaluated to false as inconsistent. For an existential quantifier formula, we record those links that have caused the formula evaluated to true as consistent. Thus, the result includes both inconsistent and consistent links. One may ignore consistent links if he is only interested in inconsistency detection. In the case where universal quantifier formulae are satisfied or existential quantifier formulae are violated, there is no interesting link for report [14].

The semantics in Figure 10 is only for full checking. We continue to discuss how to generate such links incrementally. Figure 11 gives our link generation semantics for incremental checking of universal quantifier formula. 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 formula is similar and thus omitted.

For and, or and implies formulae, generating links is more complicated. We take and formula for example to explain our idea. 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 $a \wedge b$, if a is evaluated to true and b to false, then the formula fails due to b and only due to b. In this case, b should be included in the link for this formula [16]. Such idea is one of major contributions of *xlinkit* and reasonable for and, or and implies formulae. Unfortunately, the integration of such an optimization design with checking semantics is not fully clear in *xlinkit* [14].

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

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

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

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

Figure 13. Link generation semantics for incremental checking of and formula.

We deploy this optimization design in our link generation semantics. We tightly integrate link generation with the optimization to ensure links to exclude non-core information. Then we extend it to support incremental link generation.

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

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

To study incremental link generation, we identify three cases based on affected function results and two of them are further classified into four subcases based on the above principles. Figure 13 gives a complete specification. or and implies formulae are similar and their details are thus omitted. The remaining two formula types are not and *bfunc* formulae. We consider them in Figure 14 and Figure 15. They are relatively simple (note that the not operator would flip any status).

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

5.3 Stateful Context Patterns

For effective incremental checking, interesting context instances should be retrieved without having to repetitively rescan the entire

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

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

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

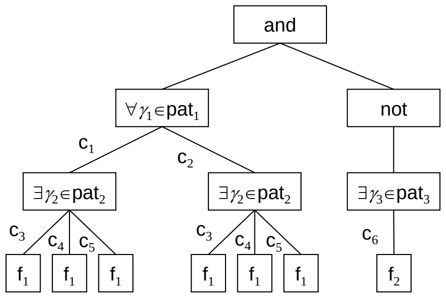

Figure 16. *CCT* of the example rule.

dataset, i.e., context history. We thus propose the notion of stateful context pattern. Context pattern is a concept similar to *xpath* selector, but they work differently. Given an *xpath* selector, one has to parse an entire *XML* document to find matched contents in terms of *DOM* tree nodes. However, this parsing is stateless in the sense that *xpath* discards the parsing result such that its potential reuse in future is destroyed. The entire document has to be parsed again next time for matched contents even if the changes to the document are minor. In contrast, each pattern, say *pat*, maintains a matching queue to store its last matched context instances (i.e., $\mathcal{M}[pat]$). Whenever a context change is identified, the change is immediately examined to decide its impact on $\mathcal{M}[pat]$. Thus, our approach avoids rescanning the context history in each checking since interesting context instances have already been in their corresponding patterns' 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 with each other, and therefore the last parsing result for an *xpath* expression is difficult to maintain in time, even if possible. In fact, even simple constraints, such as uniqueness, represented by *xlinkit* in a meta-model of *XML* documents are generally undecidable [7].

In implementation, newly detected context instances are matched against given patterns as explained earlier. Each pattern is referred to by one formula or more. Thus, the rules and sub-formulae thus affected can be easily 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 need, the rules and sub-formulae thus affected can also be determined similarly and then rechecked using the algorithm explained in the next section.

6. CHECKING ALGORITHM

In this section, we introduce consistency computation tree, and compare full checking and incremental checking for context consistency.

6.1 Consistency Computation Tree

To facilitate 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) have only 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 sizes of concerned $\mathcal{M}[pat]$ results.

$\mathcal{L}[\mathsf{not}\ (formula)]_\alpha =$

(1) $\mathcal{L}_0[\mathsf{not}\ (formula)]_\alpha$, if $\mathsf{affected}(formula) = \bot$;

(2) $\{\mathsf{flip}(l) \mid l \in \mathcal{L}[formula]_\alpha\}$, if $\mathsf{affected}(formula) = \top$.

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

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

The conversion is straightforward. Consider our aforementioned rule: $(\forall\gamma_1 \in \mathsf{pat}_1\ (\exists\gamma_2 \in \mathsf{pat}_2\ (\mathsf{f}_1(\gamma_1, \gamma_2))))$ and $(\mathsf{not}\ (\exists\gamma_3 \in \mathsf{pat}_3\ (\mathsf{f}_2(\gamma_3))))$. Suppose that $\mathcal{M}[\mathsf{pat}_1] = \{\mathsf{c}_1, \mathsf{c}_2\}$, $\mathcal{M}[\mathsf{pat}_2] = \{\mathsf{c}_3, \mathsf{c}_4, \mathsf{c}_5\}$ and $\mathcal{M}[\mathsf{pat}_3] = \{\mathsf{c}_6\}$ ($\mathsf{c}_1, \mathsf{c}_2, \ldots, \mathsf{c}_6$ represent six context instances). Then its corresponding *CCT* is as shown in Figure 16.

For a ∀ or ∃ node, each of its branches represents the sub-formula of this formula the node represents with a certain variable assignment (e.g., the leftmost branch of node $\forall\gamma_1 \in \mathsf{pat}_1$ corresponds to a variable assignment of $\{(\gamma_1, \mathsf{c}_1)\}$. The initial variable assignment for the root node is an empty set ϕ. When one traverses the *CCT* from the top down, each node may add 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 for the rule (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 require processing only small parts of the tree. We term a *critical node* as the node that contains an occurrence of the pattern affected by a given 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 of the tree. This is because they are all derived from the same sub-formula that contains the affected pattern but with different variable assignments. The checking algorithm also works in three steps: (1) adjust the *CCT* of the given rule, (2) adjust the Boolean value of the rule, and (3) adjust the links generated for the rule, as we explain below.

6.3.1 CCT Adjustment

Each critical node must be with a universal or an existential quantifier, 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 node *N*.
- **Deletion of a context instance from $\mathcal{M}[pat]$:** Delete an existing branch corresponding to *x* from node *N*.

Each critical node has to take 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 resulting *CCT* is as shown in Figure 17.

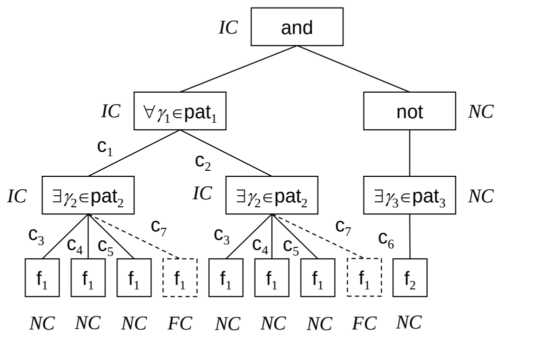
IC
and
IC
$\forall\gamma_1\in$pat$_1$
not
NC
c_1
c_2
IC
$\exists\gamma_2\in$pat$_2$
IC
$\exists\gamma_2\in$pat$_2$
$\exists\gamma_3\in$pat$_3$
NC
c_3 c_4 c_5 c_7 c_3 c_4 c_5 c_7 c_6
f_1 f_1 f_1 f_1 f_1 f_1 f_1 f_1 f_2
NC NC NC FC NC NC NC FC NC

Figure 17. *CCT* Adjustment: adding branches (dashed branches are newly added).

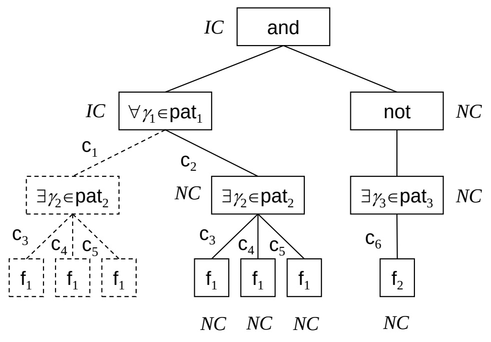
IC
and
IC
$\forall\gamma_1\in$pat$_1$
not
NC
c_1
c_2
$\exists\gamma_2\in$pat$_2$
NC
$\exists\gamma_2\in$pat$_2$
$\exists\gamma_3\in$pat$_3$
NC
c_3 c_4 c_5 c_3 c_4 c_5 c_6
f_1 f_1 f_1 f_1 f_1 f_1 f_2
NC NC NC NC

Figure 18. *CCT* Adjustment: deleting branches (the dashed branch is newly deleted).

- If $\mathcal{M}$[pat$_1$] changes from {c$_1$, c$_2$} to {c$_2$}, the resulting *CCT* is as shown in Figure 18.

6.3.2 Boolean Value and Link Adjustment

First, if the *CCT* adjustment is to add branches, all nodes on the added branches (excluding critical nodes) must be fully checked (by a post-order traversal) for Boolean values and links, since their variable assignments contain the new change.

Second, no matter 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 (by a post-order traversal) for new Boolean values and links, since each of them has at least one branch checked or rechecked.

Finally, there is no need to check or recheck other nodes.

Figure 17 and Figure 18 also annotate how to adjust Boolean values and links for the above examples (*FC*: fully checked; *IC*: incrementally checked; *NC*: no checking required).

6.4 Time Complexity Analysis

Let the node number of a *CCT* be n (including all leaf nodes and internal nodes). One can observe that a full post-order traversal of the *CCT* takes $O(n)$ time. Upon a context change, suppose that the node number of the *CCT* 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*, recalculate Boolean values and regenerate links.

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

- **Adding branches:** The time cost has two parts: (1) adjusting $n_1 - n$ nodes on the added branches (including node creation, Boolean value evaluation and link generation); (2) 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 $O(3(n_1 - n) + 2n_2) = O(3n_1 - 3n + 2n_2)$.
- **Deleting branches:** The deletion takes $O(1)$ time. The remaining 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 relationships among n, n_1 and n_2, let the distance from critical nodes to the root node be h (note that all critical nodes are at the same level of the CCT) 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 added or c existing leaf nodes deleted. Thus, $|n_1 - n| = O(c)$. When all critical nodes are the root node itself, there is a very large branch added or deleted, whose size is proportional to $O(n)$. Thus, $|n_1 - n| = O(n)$. Other cases are in between.
- $O(h + c) \leqslant n_2 \leqslant O(hc)$: The paths from critical nodes to the root node eventually merge into one path, the lowest node of which is named the *split node*. When the split node is right above critical nodes (or the split node is the only critical node itself), 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 achieved when there is only one critical node ($c = 1$), which is also the split node and right above leaf nodes. The worst case is achieved when the split node and the only critical node ($c = 1$) are the root node itself.

As a result, the best case (*FC*: full checking, *ICADD*: incremental checking – adding branches, *ICDEL*: incremental checking – deleting branches) is:

- *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 is:

- *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 much more efficient than full checking, because h is much less than n and can be considered as a constant 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 incremental checking algorithm (or *IC* for short). *FC* is actually the rule-based incremental checking discussed earlier. We used our own implementation. The reason we did not convert context instances into the *XML* format to compare our approach with *xlinkit* directly is that the conversion is time-consuming itself, compared to actual checking time, and after conversion, one cannot make use of our stateful pattern mechanism, defying the comparison purpose.

The experiments were conducted on a Pentium IV @1.3GHz machine running Microsoft Windows XP Professional SP2. The test

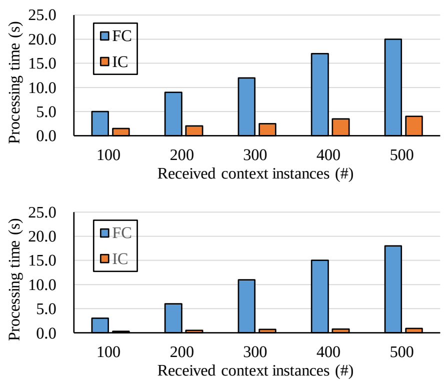

Figure 19. Performance comparison between *FC* and *IC* (top: *ADD*, bottom: *DEL*, time interval: 0.2s).

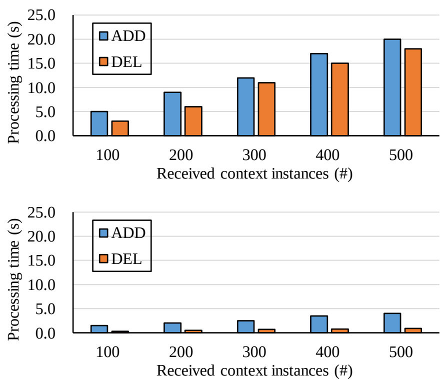

Figure 20. Performance comparison between *ADD* and *DEL* (top: *FC*, bottom: *IC*, time interval: 0.2s).

environment consists of two parts: middleware and client. A simulated context source (client) fed new context instances to the middleware at a controlled rate. To study the impact of different workloads, we gradually increased the time interval between two consecutive context instances from 0.1 to 0.4 seconds at a pace of 0.1 seconds. We also collected experimental results when the total number of context instances varied from 100 to 500 at a pace of 100. Note that the number of context changes is twice as many as that of generated context instances since each context instance was processed by the middleware twice: once as a context addition change (or *ADD* for short) and once as a context deletion change (or *DEL* for short) due to its expiration some time later. We fixed the freshness need to be five seconds, meaning that each context instance remained valid for five seconds, during which it could participate in any rule checking. We chose *Cabot* [24] as our middleware infrastructure to incorporate our context service. Our context consistency checking worked 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* discussed earlier has only three levels). The rules cover all seven *FOL* formula types. Every rule contains two universal or existential quantifier sub-formulae since they would cause considerable 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 was matched, it activated the checking of relevant rules that were built on it. Among all context instances, 66.7% of them contributed to context inconsistency, i.e., they caused the inconsistent status during checking.

Figure 19 compares the performance between *FC* and *IC* when the time interval was fixed to 0.2 seconds (top: the *ADD* scenario, bottom: the *DEL* scenario). Both sub-figures show that *IC* is superior to *FC*. *IC* is approximately five times faster than *FC* for the *ADD* scenario and even faster for the *DEL* scenario.

Figure 20 compares the performance between the *ADD* and *DEL* scenarios when the time interval was fixed to 0.2 seconds (top: *FC*, bottom: *IC*). Two results are quite different. For *FC*, its processing time for the *ADD* and *DEL* scenarios is quite close to each other; but for *IC*, there is a large difference. It is understandable since from the earlier analysis, *FC*'s time complexity is $O(n)$ for both the *ADD* and *DEL* scenarios, but *IC*'s time complexity is between $O(h)$ and $O(n)$ for the *ADD* scenario and always $O(h)$ for the *DEL* scenario. In practice, h is much less than n. Thus, *IC* has an even greater advantage over *FC* for the *DEL* scenario. The experimental

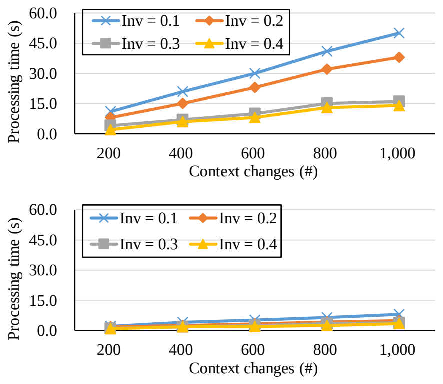

Figure 21. Performance comparison under different workloads (top: *FC*, bottom: *IC*).

results confirm our earlier analysis.

Figure 21 compares the performance under different workloads by changing the time interval from 0.1 to 0.4 seconds (top: *FC*, bottom: *IC*). Although two algorithms' performance differs a lot (about seven times), their trends behave similarly: when the time interval was decreased, the processing time was increased accordingly. This is because we fixed the freshness need in experiments, and when one has more context instances fed to the middleware during a time unit (i.e., a smaller time interval), there would be more context instances valid for consistency checking (i.e., longer matching queues). Thus, the processing time must be increased since all valid context instances have to be examined.

The above experimental results suggest that *IC* is more suitable than *FC* for dynamic environments, where context changes are frequent. Currently, the results are subject to our own implementation, and more experiments would be necessary on other implementations to conclude whether they follow similar trends as we report in this paper.

8. CONCLUSION

In this paper, we have proposed a novel formula-based incremental checking approach for context consistency in pervasive computing. Our approach carefully distinguishes the core part from non-core part in checking to improve the performance. The core part rechecks affected sub-formulae since the incurred rechecking is inevitable. The non-core part reuses previously checked and still valid results to substitute the effort of rechecking unaffected sub-formulae.

Our approach has been heavily influenced by the *xlinkit* technology. The notation of link is from *xlinkit*, but we use a different approach to generating them. The idea of auxiliary function is also adapted from *xlinkit*. The difference lies in that their original uses are for rule-based incremental checking, while we have adapted them to our incremental checking with a finer granularity. *xlinkit* is stateless among consecutive checkings of the same formula, for which our context pattern mechanism is stateful, and thus we realize reusing stateful information in our approach.

We have also studied the intrinsic imperfectness of pervasive contexts, and identified a mandate of context consistency checking. We have proposed formal semantics for incrementally checking *FOL* 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 the *Cabot* middleware. The experiments have validated our effort.

Currently, our link generation still suffers 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]) can be 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 later checking. We will consider how to alleviate the state explosion problem in future, which can be serious when matching queues are very long. Currently, the use of rules is limited to expressing necessary conditions for context consistency, which can be easily extended for expressing sufficient conditions for context inconsistency. Besides, we plan to apply our incremental checking to situation assessment, which is critical to context-aware applications since they need to respond quickly to environmental changes.

ACKNOWLEDGMENTS

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

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. In *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. In *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. In *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. In *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. In *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. In *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. In *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. In *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. In *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, In *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. In *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. In *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. In *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. In *Proceedings of the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 336–345, Lisbon, Portugal, Sep 2005.

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