Detection of Behavioral Contextual Properties in Asynchronous Pervasive Computing Environments

Yu Huang\textsuperscript{1,2}, Jianping Yu\textsuperscript{1,2}, Jiannong Cao\textsuperscript{3}, Xianping Tao\textsuperscript{1,2}
\textsuperscript{1}State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China
\textsuperscript{2}Department of Computer Science and Technology, Nanjing University, Nanjing 210093, China
\{yuhuang, txp\}@nju.edu.cn, csjpyu@gmail.com
\textsuperscript{3}Internet and Mobile Computing Lab, Department of Computing, Hong Kong Polytechnic University, Hong Kong, China
csjcao@comp.polyu.edu.hk

Abstract—Detection of contextual properties is one of the primary approaches to enabling context-awareness. In order to adapt to temporal evolution of the pervasive computing environment, context-aware applications often need to detect behavioral properties specified over the contexts. This problem is challenging mainly due to the intrinsic asynchrony of pervasive computing environments. However, existing schemes implicitly assume the availability of a global clock or synchronous coordination, thus not working in asynchronous environments. We argue that in pervasive computing environments, the concept of time needs to be reexamined. Toward this objective, we propose the Ordering Global Activity (OGA) algorithm, which detects behavioral contextual properties in asynchronous environments. The essence of our approach is to utilize the message causality and its on-the-fly coding as logical vector clocks. The OGA algorithm is implemented and evaluated based on the open-source context-aware middleware MIPA. The evaluation results show the impact of asynchrony on the detection of contextual properties, which justifies the primary motivation of our work. They also show that OGA can achieve accurate detection of contextual properties in dynamic pervasive computing environments.

I. INTRODUCTION

Pervasive applications are typically context-aware, using various kinds of contexts, such as location and time, to provide smart services [1], [2]. Context-aware applications need to detect whether contexts bear specified properties, thus being able to adapt to the pervasive computing environment accordingly [3], [4].

Detection of contextual properties has been widely studied in pervasive computing and software engineering communities [5], [6], [7], [8], [9], [10]. For example in [6], contextual properties are expressed in first order logic, and properties such as “location of the user is the meetingroom and a presentation is going on” can be specified. However, existing schemes mainly focus on the detection of static properties, i.e. properties at given snapshot of time. Though static properties can capture interesting aspects of the pervasive computing environment, they inherently lack the notions of relative temporal order [11], [12]. Such properties cannot characterize temporal evolution of the environment, such as “\textit{C}_1$: the user is in his office, and then the user leaves the office”.

The discussions above necessitate the detection of behavioral properties, i.e. properties delineating behavior patterns or temporal evolution of the contexts. The key issue in detection of behavioral properties is how to decide the temporal order among contextual activities. This issue is challenging mainly due to the following two observations:

- Contextual activities are often global, involving multiple decentralized context collecting devices. For example, in property \textit{C}_1 discussed above, the location context is decided by two different sensors (RFID reader and light sensor), in order to improve the accuracy of context. Pervasive applications and context collecting devices usually coordinate in a fully-distributed manner, based on wired/wireless communications.

- The pervasive computing environment is often asynchronous [13], [10], [12]. Specifically, context collecting devices do not necessarily have a global clock. They heavily rely on wireless communications, which suffer from bounded but arbitrary delay. Moreover, due to resource constraints, context collecting devices, e.g. battery-powered sensors, often need to buffer context data for certain time [13]. The different context update rates also result in the asynchrony of pervasive computing environments. However, existing schemes implicitly assume that the contexts being checked belong to the same snapshot of time [5], [6], [7], [8]. This assumption does not necessarily hold in pervasive computing environments.

To address the challenges above, we study in this paper the checking of behavioral contextual properties in asynchronous environments. Specifically,

- We define behavioral properties based on the ordering of global contextual activities. We first define global activities based on the concurrency among multiple local activities. Then, both the concurrency among local activities and the relative order among global activities are defined based on the happen-before relation resulting from the message causality [14].

- We propose the Ordering Global Activities (OGA) al-
algorithm to check behavioral contextual properties. OGA is implemented based on one of our research projects Middleware Infrastructure for Predicate detection in Asynchronous environments (MIPA) [15].

- The OGA algorithm is evaluated in a smart-lock scenario, which is first investigated in our previous work [16]. The evaluation results show how the asynchrony in the pervasive computing environment affects the detection of behavioral properties. The results also show the accuracy of OGA in detection of contextual properties.

The rest of this paper is organized as follows. In Section II, we describe our system model. In Section III, we present the design of the OGA algorithm. In Section IV and V, we overview the design of MIPA and present the experimental evaluation. Section VI overviews the existing work. In Section VII, we conclude the paper with a brief summary and the future work.

II. SYSTEM MODEL

In this section, we first discuss how we model asynchronous pervasive computing environments. Then we discuss how to specify behavioral contextual properties, which includes specification of global activities and specification of the relative order among global activities. Notations used in the system model are listed in Table I.

Detection of contextual properties assumes the availability of an underlying context-aware middleware. The middleware accepts contextual properties specified by the application, detects at runtime whether the specified properties hold, and informs the application of the results (see more discussions in Section IV). Specifically, a collection of non-checker processes \( P_1, P_2, \ldots, P_n \) \(^1\) are deployed to monitor specific regions of the environment. Examples of non-checker processes are sensor agents manipulating physical sensors. One checker process \( P_{che} \) collects context information from non-checker processes, and achieves runtime detection of contextual properties. \( P_{che} \) is usually a third-party service deployed on the context-aware middleware.

A. Asynchronous Pervasive Computing Environments

We model context-aware applications in asynchronous pervasive computing environments as a loosely coupled message-passing system, without any global clock or shared memory. Communications suffer from uncertain delay. Dissemination of context data may be postponed due to resource constraints. We assume that no messages are lost, altered, or spuriously introduced. We do not assume that the underlying communication channel is first-in-first-out (FIFO). Justifications for the assumptions are discussed in Section III.D.

Detection of contextual properties is based on the classical Lamport’s definition of the happen-before (denoted by \( \rightarrow \)) relation resulting from message causality [14] and its “on the fly” coding given by Mattern and Fidge’s vector clocks [17], [18]. Each non-checker process \( P_i \) keeps \( VC_j \), its own vector clock timestamps. \( VC_j[i](i \neq j) \) is ID of the last message from \( P_j \), which has a causal relation to \( P_i \). \( VC_j[j] \) for \( P_j \) is the next message ID \( P_j \) will use. Messages passed in the system can be classified into two types:

- **Control message.** Non-checker processes send control messages among each other to establish the happen-before relation among contextual activities.
- **Checking message.** Non-checker processes send vector clock timestamps of contextual activities via checking messages to \( P_{che} \). \( P_{che} \) decides whether specified property holds based on the collected timestamps.

B. Global Activities

Contextual activities can be either local or global. A local activity takes place on some \( P_i \) without any interaction with other processes. We delineate local activities of our concern on non-checker process \( P_i \) with local predicate \( LA_i \). \( LA_i \) is true if the local activity is taking place on \( P_i \). Otherwise, it is false. We record the interval in which \( LA_i = true \). The false-to-true and the true-to-false transitions (denoted by \( \uparrow \) and \( \downarrow \) respectively) of \( LA_i \) correspond to the beginning and ending of the interval, which are denoted by \( I_i.lo \) and \( I_i.hi \) respectively.

A global activity results from the interaction among local activities. The interaction projected on the time axis is the concurrency among local activities, i.e., the overlapping of intervals of local activities. To detect whether \( I_1, I_2, \ldots, I_n \) overlap, we need to check whether the following Formula (1) is satisfied:

\[
(I_j.lo \rightarrow I_k.hi) \wedge (I_k.lo \rightarrow I_j.hi), \forall 1 \leq j \neq k \leq n
\]  

(1)

The case of three concurrent local activities is shown in Fig. 1. Detection of concurrent activities has been studied in [19], as well as in our previous work [10].

We can further classify global activities based on how we care about the time scope of the interaction, i.e., how we define the interval of the global activity. Specifically, we can define

<table>
<thead>
<tr>
<th>Notation</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>number of all non-checker processes</td>
</tr>
<tr>
<td>( m )</td>
<td>number of global activities</td>
</tr>
<tr>
<td>( GA_i )</td>
<td>the ( k^{th} ) global activity ((1 \leq k \leq m)), which might be ( GA_{i,i} ) or ( GA_{i,j} )</td>
</tr>
<tr>
<td>( size(GA_i) )</td>
<td>number of non-checker processes involved in ( GA_i )</td>
</tr>
<tr>
<td>( P_i )</td>
<td>the ( i^{th} ) non-checker process , ( 1 \leq i \leq n )</td>
</tr>
<tr>
<td>( P^{(k,j)} )</td>
<td>the ( j^{th} ) non-checker process in ( GA_k ), ( 1 \leq j \leq size(GA_k) )</td>
</tr>
<tr>
<td>( VC_i )</td>
<td>vector clock timestamp on ( P_i )</td>
</tr>
<tr>
<td>( LA_i )</td>
<td>the ( i^{th} ) local activity</td>
</tr>
<tr>
<td>( LA^{(k,j)} )</td>
<td>the ( j^{th} ) local activity involved in ( GA_k ) on ( P^{(k,j)} )</td>
</tr>
<tr>
<td>( I(GA), I(LA) )</td>
<td>interval of a global / local activity</td>
</tr>
</tbody>
</table>

\(^1\)When discussing global activities later, we use another notation \( P^{(k,j)} \) for the same process. See more explanations in Table I.
1. Concurrent local activities

\[ \bigwedge \cdot \cdot \cdot \bigwedge \text{LA} \bigvee \cdot \cdot \cdot \bigvee \text{LA} : \text{LA} \]

An or-activity takes place in the whole period \([GA] = \bigwedge_{1 \leq i \leq \text{size}(GA_k)} I(LA_i)\) of overlapping local activities. For and-activity \(GA^{\text{and}}_k = \bigwedge_{1 \leq i \leq \text{size}(GA_k)} I(LA_i)\), its interval is defined as:

\[ I(GA^{\text{and}}_k) = \bigcap_{1 \leq i \leq \text{size}(GA_k)} I(LA_i) \]

For example in Fig. 2, if we define \(GA' = LA_1 \lor LA_2\), we have that:

\[ I(GA') = I_1 \cup I_2 = [I_1\text{.lo}, I_2\text{.hi}] \]

2. Intervals for and- and or-activities

1) And-activity: An and-activity takes place in the period in which multiple local activities are interacting with each other. For example, “Alice and Bob are in the meeting room” is an and-activity. It takes place in the period when Alice and Bob are both in the meeting room. The interval of an and-activity is defined as the intersection among the intervals of overlapping local activities. For and-activity \(GA^{\text{and}}_k = \bigwedge_{1 \leq i \leq \text{size}(GA_k)} I(LA_i)\), its interval is:

\[ I(GA^{\text{and}}_k) = \bigcap_{1 \leq i \leq \text{size}(GA_k)} I(LA_i) \]

For example in Fig. 2, \(GA = LA_1 \land LA_2\). Based on the happen-before relation established, we have that:

\[ I(GA) = I_1 \cap I_2 = [I_2\text{.lo}, I_1\text{.hi}] \]

2) Or-activity: An or-activity takes place in the whole period of interaction, i.e., from the happening of the first local activity to the ending of the last local activity. For example, imagine that Alice first waits for Bob in the meeting room. When Bob comes, they have discussions. Then Alice leaves the meeting room. In this case, the or-activity “Alice or Bob is in the meeting room” takes place in the period starting from the time Alice enters the meeting room and ending at the time Bob leaves. The interval of an or-activity is defined as union of the intervals of overlapping local activities. For or-activity \(GA^{\text{or}}_k = \bigvee_{1 \leq i \leq \text{size}(GA_k)} I(LA_i)\), its interval is defined as:

\[ I(GA^{\text{or}}_k) = \bigcup_{1 \leq i \leq \text{size}(GA_k)} I(LA_i) \]

For example in Fig. 2, if we define \(GA' = LA_1 \lor LA_2\), we have that:

\[ I(GA') = I_1 \cup I_2 = [I_1\text{.lo}, I_2\text{.hi}] \]

C. Ordering Global Activities

Due to the distributed nature of contexts, we often rely on global activities to delineate the static properties of contexts. To delineate the behavioral patterns of contexts, applications are interested in (global) activities which take place in specified temporal order, such as “GA_1 happens, then GA_2 happens, ..., finally GA_m happens”. For example, in the behavioral property C_1 discussed in Section I, the application is interested in the relative order between two global activities “the user is in the office” and “the user is in the corridor (leaves the office)”. A sequence of ordered global activities is defined as:

\[ S_{GA} : = GA_1 \prec GA_2 \prec \cdots \prec GA_m \]

Here, \(GA_k\) proceeds \(GA_{k+1}\) is defined as the happen-before relation between the corresponding intervals:

\[ GA_k \prec GA_{k+1} : = I(GA_k).hi \rightarrow I(GA_{k+1}).lo \]

In the next section, we discuss how to check the ordering of global activities in asynchronous pervasive computing environments.

III. ORDERING GLOBAL ACTIVITIES IN ASYNCHRONOUS PERVERSIVE COMPUTING ENVIRONMENTS

In this section, we present design of the proposed Ordering Global Activities (OGA) algorithm. The OGA algorithm consists of three parts: 1) the non-checker process specifies the message activities upon changes in the local predicate value; 2) the checker process first detects global activities; 3) then the checker process builds the ordering among global activities. Notations used in the design of OGA are listed in Table I and II.

A. Message Activities on Non-checker Process \(P^{(k,t)}\) in \(GA_k\)

On the non-checker process \(P^{(k,t)}\), different message activities are specified upon the beginning and ending of the local activity:

- Upon \(LA^{(k,t)}\), a control message is sent to every \(P^{(k,s)}(1 \leq s \leq \text{size}(GA_k), s \neq t)\), i.e., all other non-checker processes in the same global activity with \(P^{(k,t)}\). The message activity here aims at building the happen-before relation required in Equation (1), in order to detect \(GA_k\).
Upon $LA^{(k,t)}\downarrow$, a control message is sent among every other non-checker processes $P_i (P_i \neq P^{(k,t)})$. The message activity here aims at ordering different global activities, as required in Equation (2). Meanwhile, a checking message is sent to $P_{che}$. This checking message sends vector clock timestamps $([lo, hi])$ of $I(LA^{(k,t)})$ to $P_{che}$ for the detection and ordering of global activities, as discussed in Section III.B and III.C respectively.

Boolean variable $flagMsgAct$ is used to reduce redundant message passing, as in [19], [10]. The initial value of $flagMsgAct$ is true. Pseudo codes of OGA on the non-checker process side are listed in Algorithm 1.

B. Detecting Global Activities

1) Checking the concurrency: Checking messages from all the non-checker processes are grouped according to the global activity they belong to. For given global activity $GA_k$, we check the concurrency among local activities based on Formula (1). The checker process has a separate queue $Que^{(k,t)}$ for each $P^{(k,t)}$ in $GA_k$. Incoming checking messages are enqueued in the appropriate queue.

We assume that $P_{che}$ receives messages from each non-checker process in FIFO order as in [20], [19]. Note that this is not a restrictive assumption. We do not require FIFO for the underlying communication. $P_{che}$ needs to implement the FIFO property for efficiency purposes. If the underlying communication is not FIFO, $P_{che}$ ensures this property by using sequence numbers in messages.

Each element of $Que^{(k,t)}$ is timestamp $[lo, hi]$ of an interval. The $lo$ and $hi$ are compared to check the concurrency among intervals. The checker process reduces the number of comparisons by deleting any interval at the head of any queue whose $hi$ is not greater than $lo$ of the interval at the head of all other queues. $P_{che}$ detects $GA_k$ if it finds a set of intervals at the head of queues such that they are pairwise overlapping. The detection of concurrency is mainly based on the strong conjunctive predicate algorithm in [19] and our previous work [10].

2) Calculating the interval of $GA_k$: After the detection of $GA_k$, we need to calculate $I(GA_k)$, the interval of this global activity, for further ordering of global activities. For and-activities, we need to calculate the intersection of intervals, while for or-activities, we need to calculate the union of intervals, as shown in Fig. 2.

In the ideal case, for an and-activity, we calculate the latest $lo$ (every other $lo$ happens before it) and the earliest $hi$ (happening before every other $hi$). However, we may not always be able to obtain the latest/earliest $lo/hi$ in asynchronous environments. For example in Fig. 3, we cannot decide which one is later for $I_2.lo$ and $I_3.lo$. Neither can we decide which one is earlier for $I_1.hi$ and $I_3.hi$. Thus, for all the $los$, we prune those which happens before any other $lo$ (must not be the latest), and keep all the remaining (concurrent) $los$. Similarly, for all the $his$, we prune those which “happens after” any other $hi$ (must not be the earliest), and keep all the remaining (concurrent) $his$. For example in Fig. 3, we need to keep $I_2.lo$ and $I_3.lo$, as well as $I_1.hi$ and $I_3.hi$.

The or-activity is the dual of and-activity. Similar duality remains in calculating the interval of and- or-activities.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$CurIntv$</td>
<td>interval of local activity on the non-checker process</td>
</tr>
<tr>
<td>$flagMsgAct$</td>
<td>boolean value used to denote whether there have been new message activities</td>
</tr>
<tr>
<td>$VC^{(k,t)}$</td>
<td>vector clock timestamp on $P^{(k,t)}$</td>
</tr>
<tr>
<td>$Que^{(k,t)}$</td>
<td>queue for $P^{(k,t)}$ in $GA_k$ on the checker process</td>
</tr>
<tr>
<td>$QueLo_k,QueHi_k$</td>
<td>queues for recording results of detecting $GA_k$</td>
</tr>
<tr>
<td>$CurQueLo, CurQueHi$</td>
<td>current global activity to be ordered previous global activity which has been ordered</td>
</tr>
</tbody>
</table>

**Algorithm 1 OGA on $P^{(k,t)}$ in $GA_k$**

1: Upon $LA^{(k,t)}\uparrow$
2: send$_{control}(VC^{(k,t)})$ to each $P^{(k,s)}$ in $GA_k$, $s \neq t$;
3: if $flagMsgAct$ then
4: $CurIntv.lo := VC^{(k,t)}$;
5: end if

6: Upon $LA^{(k,t)} \downarrow$
7: send$_{control}(VC^{(k,t)})$ to each $P_i (1 \leq i \leq n, P_i \neq P^{(k,t)})$;
8: if $flagMsgAct$ then
9: $CurIntv.hi := VC^{(k,t)}$;
10: send$_{checking}(CurIntv[lo, hi])$ to $P_{che}$;
11: $flagMsgAct := false$;
12: end if

13: Upon receive$_{control}$$_{msg}(VC_i)$ from $P_i$
14: for $j = 1$ to $n$ do
15: $VC^{(k,t)}[j] := max\{VC^{(k,t)}[j], VC_i[j]\}$;
16: end for
17: $flagMsgAct := true$;

![Fig. 3. Calculating the interval of a global activity](image-url)
Algorithm 2 Detecting $GA_k$ in OGA

1: Upon receiving $CurIntv[p, h]$ from $P^{(k,i)}$
2: insert $CurIntv[p, h]$ to $Que^{(k,i)}$;
3: if $CurIntv[p, h] \neq Que^{(k,i)}$.head() then
4: return;
5: end if
6: while $changed \neq \phi$ do
7: for each $P^{(k,i)}$ in $changed$ and $P^{(k,j)}$ in $GA_k$ do
8: if $Que^{(k,j)}$.head().lo $\neq$ $Que^{(k,i)}$.head().hi then
9: $newchanged := newchanged \cup \{P^{(k,i)}\}$;
10: end if
11: $ Que^{(k,j)}$.head().lo $\neq$ $Que^{(k,i)}$.head().hi then
12: $newchanged := newchanged \cup \{P^{(k,j)}\}$;
13: end if
14: end for
15: $changed := newchanged$;
16: for each $P^{(k,i)}$ in $changed$ do
17: delete_head($Que^{(k,i)}$);
18: end for
19: if $GA_k$ is detected */
20: if $\forall i, Que^{(k,i)}$ is not empty then
21: calculate $I(GA_k)$;
22: enqueue each $lo$ and $hi$ remained after the pruning to $QueLo(GA_k)$ and $QueHi(GA_k)$ respectively;
23: end if

For an or-activity, we need the earliest $lo$ and the latest $hi$.

Pseudo codes for the detection of global activities are listed in Algorithm 2.

C. Ordering Global Activities

The essential issue in ordering two global activities is to establish the relative order between $I(GA_k).hi$ and $I(GA_{k+1}).lo$. As discussed in the previous section, we may encounter multiple (concurrent) $los$ and $his$ when detecting global activities. We have stored all these $los$ and $his$ in appropriate queues as shown in Algorithm 2. Now, we compare all the stored $los$ and $his$. This comparison continues until $I(GA_k).hi \rightarrow I(GA_{k+1}).lo$ is established for every stored $hi$ and $lo$. When we reach the last global activity, we finish the ordering of global activities. Pseudo codes for the ordering of global activities are listed in Algorithm 3.

D. Discussions

The number of comparisons for detecting a global activity is $O(s^2p)$, where $s$ is the upper bound of size of the global activity, $p$ is the upper bound of length for each queue in detecting the global activity. The number of comparisons for ordering global activities is $O(s^2m)$. On the normal process side, the number of message activities is $O(p)$. Note that existing work may impose less message complexity, but they rely on the assumption of a global clock or synchronized interactions. The message complexity of OGA is mainly due to building the happen-before relation between $los$ and $his$, which is a requisite for detecting temporal properties in asynchronous environments.

We assumed reliability of message passing. Note that even with this assumption, we cannot guarantee correct ordering of global activities. Without this assumption, we only need to revise our algorithm to tolerate incomplete message information. Rationale of our algorithm remains the same. The probability of detecting global activities is analyzed in our previous work [10]. In Section V, we further evaluate OGA by experiments.

IV. Implementation

The OGA algorithm we propose assumes the availability of an underlying context-aware middleware. We have developed such a middleware named Middleware Infrastructure for Predicate detection in Asynchronous environments (MIPA) [15]. From MIPA’s point of view, a pervasive computing environment is composed of an application layer, a middleware layer and a context source layer, as shown in Fig. 4.

The middleware layer is the kernel part of MIPA. Its fundamental functionalities include:

- **Property detection manager.** The predicate broker accepts contextual properties specified by the context-aware application. It first parses the property, and then initiates the non-checker processes and the checker process accordingly.
- **Non-checker process.** The non-checker process monitors the local predicate value based on the Event-Condition-Action (ECA) mechanism. The non-checker process sends messages to build the requisite happen-before relation. It also sends checking message to the checker process, which finally decides whether the contextual property holds.
- **Checker process.** The checker process collects vector clock timestamps of local contextual activities. It decides whether the application-specified contextual property holds, and informs the application.

We implement the OGA algorithm on MIPA, and conduct the experimental evaluation, as discussed in detail in the next
section.

V. EXPERIMENTAL EVALUATION

In the previous Section III, we presented design of the OGA algorithm. However, does OGA work in pervasive computing environments? Specifically, can OGA achieve accurate ordering of global activities? We investigate these issues by experiments in this section.

A. Experiment setup

The experimental evaluation is based on a smart-lock scenario first investigated in our previous work [16]. In this scenario, a smart-lock application automatically locks the office when the users leaves, i.e., when user’s location changes from ‘office’ to ‘corridor’. To deal with noisy sensor readings, the user’s location context is detected by both an RFID reader and a light sensor. User’s location is detected by the global activity \( GA_1 = (\text{the user is detected by the RFID reader in the office}) \land (\text{the user is detected by the light sensor in the office}) \) and \( GA_2 = (\text{the user is detected by the RFID reader in the corridor}) \land (\text{the user is detected by the light sensor in the corridor}) \). User’s behavior of leaving the office is delineated by the behavioral property \( GA_1 \prec GA_2 \).

We model user’s stay in the office based on the queueing theory [10]. Specifically, a queue of intervals with Poisson arrival rate \( \frac{1}{600} \) is adopted. The duration of intervals follows the exponential distribution of rate \( \frac{1}{300} \). We model the message delay by the exponential distribution. Note that the distribution of message delay is affected by implementation of the underlying network layers (e.g., the MAC or routing layer), and greatly varies in different scenarios. Though it is doubted whether there exists a universal model of message delay, the exponential distribution is widely used and evaluated by both simulations and experiments [21]. Our experiment methodology is also applicable when the message delay follows other types of distributions.

In the evaluation, we study how asynchrony of the computing environment affects the performance of OGA. The update interval of sensor data dissemination and the message delay are varied. This issue is critical since the asynchrony is the primary motivation of our work. We also study the effect of tuning the duration of the user’s stay in the office. This duration decides how frequently the user leaves the office.

Performance of the OGA algorithm is measured by the probability of correct ordering of global activities in asynchronous environments. We obtain this probability of correct ordering by calculating the ratio of \( \frac{\text{Num}_{\text{OGA}}}{\text{Num}_{\text{phy}}} \). Here, \( \text{Num}_{\text{OGA}} \) denotes how many times OGA detects the ordering of global activities. \( \text{Num}_{\text{phy}} \) denotes the number of the ordering of global activities, obtained from physical time of each local contextual activity. Detailed experiment configurations are listed in Table III.

B. Effects of Tuning the Update Interval

In this experiment, we study the effect of tuning length of the update interval of the sensors. We find that the increase in the update interval results in monotonic decrease in the probability of correct ordering of global activities, as shown in Fig. 5 and 6. The is mainly because the increase of update interval adds to the asynchrony of the environment. Specifically, the probability of correct ordering is high (over 90%) when the update interval is less than 10 minutes, as shown in Fig. 5. When the update interval gets longer than the average duration of the user’s stay in the office (10 minutes), the probability begins to decrease much more quickly, as shown in Fig. 5. When we increase the update interval to a large value (up to 90 minutes), the probability may decrease to around 20%, as shown in Fig. 6.

In summary, the evaluation results here show the impact of asynchrony of the environment on the checking of behavioral properties. They also show that OGA can achieve accurate checking, even when the update interval is reasonably long.
C. Effects of Tuning the Message Delay

In this experiment, we study how the message delay affects the performance of OGA. We find that when encountered with reasonably long message delay (less than 1s), the probability of correct detection is quite high (a little less than 100%), as shown in Fig. 7. Only when the delay goes up to more than 1 minute, the probability begins to significantly decrease, as shown in Fig. 8. Note that though the message delay usually does not go up to several minutes, we increase the message delay to large values here to explore its impact on the performance of OGA.

Combining the results in Fig. 7 and 8, we also find that the message delay results in monotonic decrease in the probability of correct ordering of global activities, mainly due to the increase in the asynchrony of the environment. However, the impact of the message delay is comparatively less than that of the update interval.

D. Effects of tuning the Duration of User’s Stay in the Office

In this experiment, we tune the duration of user’s stay in the office. We find that tuning the duration does not has as much impact as that of tuning the update interval and the message delay, as shown in Fig. 9. The probability of correct detection slowly decreases as the duration increases. The probability first decreases as the duration increases to 15 minutes. Then it remains relatively stable. The probability decreases again when the duration increases to 50 minutes.

The duration of stay does not affect the asynchrony of the environment, thus imposing less impact on the performance of OGA. The probability of correct detection slightly decreases mainly because when the duration of stay increases, the user leaves the office less frequently. The number of global activities which can be ordered by OGA decreases.

E. Lessons Learned

Based on the experimental evaluation, we first show the impact of asynchrony in the pervasive computing environment on detection of contextual properties, which justifies the basic motivation of our work. We also demonstrate the performance of OGA in pervasive scenarios. Specifically, OGA achieves high probability of ordering global activities, even when faced with reasonably long update interval and message delay, as well as when faced with different frequencies of contextual behaviors.

VI. RELATED WORK

Many existing studies on context-aware computing are concerned with middleware infrastructures that support collection
and management of contexts [3], [1], [4]. Various schemes have been proposed for detection of contextual properties over context-aware middleware. In [5], contextual properties were modeled by tuples, and property detection was based on comparison among elements in the tuples. In [6], properties were expressed in first-order logic, and an incremental detection algorithm was proposed. In [9], a probabilistic approach is proposed to further improve the effectiveness of detection of contextual properties. In [7], [8], contextual properties were expressed by assertions. However, existing schemes do not sufficiently consider the temporal relation among the contexts. It is implicitly assumed that the contexts being checked belong to the same snapshot of time. Such limitations make these schemes do not work in asynchronous pervasive computing environments [13], [10], [12].

In asynchronous environments, the concept of temporal ordering of events must be carefully reexamined [14]. The happen-before relation intrinsic in message passing is a promising solution to detection of contextual properties in asynchronous pervasive computing environments. In our previous work [10], the Concurrent Event Detection for Asynchronous consistency checking (CEDA) algorithm was proposed to detect concurrent contextual activities in asynchronous environments. CEDA explicitly checks whether contexts being checked belong to the same snapshot of time based on the happen-before relation among the beginning and ending of contextual activities. However, behavior patterns of contexts cannot be specified and checked based on CEDA. In this paper, we study how to check behavioral patterns of contexts based on the ordering of global contextual activities.

VII. CONCLUSION AND FUTURE WORK

In this paper, we study how to check behavioral properties of contexts in asynchronous pervasive computing environments. Toward this objective, our contribution is three-fold: 1) we delineate behavioral properties of contexts based on the ordering of global activities; 2) we propose the OGA algorithm to check behavioral properties; 3) we implement the context-aware middleware MIPA. The OGA algorithm is implemented and evaluated over MIPA.

In our future work, we will study the design of a general framework, covering various existing predicates, as well as their checking algorithms. The framework will help us better understand the pervasive computing environment from a predicate detection perspective. We will also extend MIPA to support the proposed framework.

ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China (No. 60903024, 60736015, 60721002), the National 973 Program of China (2009CB320702), and the National 863 Program (2009AA01Z117).

REFERENCES