Runtime Detection of the Concurrency Property in Asynchronous Pervasive Computing Environments

Yu Huang, *Member*, *IEEE*, Yiling Yang, Jiannong Cao, *Senior Member*, *IEEE*, Xiaoxing Ma, *Member*, *IEEE*, Xianping Tao, *Member*, *IEEE*, and Jian Lu

Abstract—Runtime detection of contextual properties is one of the primary approaches to enabling context-awareness in pervasive computing scenarios. Among various properties the applications may specify, the concurrency property, i.e., property delineating concurrency among contextual activities, is of great importance. It is because the concurrency property is one of the most frequently specified properties by context-aware applications. Moreover, the concurrency property serves as the basis for specification of many other properties. Existing schemes implicitly assume that context collecting devices share the same notion of time. Thus, the concurrency property can be easily detected. However, this assumption does not necessarily hold in pervasive computing environments, which are characterized by the asynchronous coordination among heterogeneous computing entities. To cope with this challenge, we identify and address three essential issues. First, we introduce logical time to model behavior of the asynchronous pervasive computing environment. Second, we propose the logic for specification of the concurrency property. Third, we propose the Concurrent contextual Activity Detection in Asynchronous environments (CADA) algorithm, which achieves runtime detection of the concurrency property. Performance analysis and experimental evaluation show that CADA effectively detects the concurrency property in asynchronous pervasive computing scenarios.

Index Terms—Concurrency property, context-awareness, asynchronous environment.

1 Introduction

Pervasive computing creates environments that embed computation and communication in a way that organically interacts with humans to ease their daily behavior [1]. Contexts refer to the pieces of information that capture the characteristics of computing environments, and context-awareness allows applications to dynamically adapt to the environment [2], [3], [4].

Context-aware applications need to detect whether contexts bear specified properties, in order to adapt their behavior accordingly [3], [4], [5]. For example, an smart phone application may specify property C_1: *location of Bob is the meeting room and a presentation is going on in this room*. Thus, the application can adaptively turn the phone to silent mode when C_1 holds. Moreover, contexts are often error-prone due to noises [6]. Users may specify properties which accurate contexts must obey, based on their understanding of physical laws. Thus, noisy contexts violating such properties can be eliminated [3], [4]. For example, the application may specify that C_2: *Bob cannot appear in both Room A and Room B*, to make sure that the location context is accurately collected.

Among various contextual properties the application may specify, the *concurrency property*, i.e., property delineating concurrency among contextual activities, is of great importance. It is because the concurrency property is one of the most frequently specified properties (e.g., in [7], [3], [4]). Moreover, the concurrency property serves as the basis for the specification of many other properties. Refer to the examples above. Property C_1 implicitly depends on the concurrency between two contextual activities: *Bob is in the meeting room* and *a presentation is going on in the room*. Similarly, C_2 also depends on the concurrency between involved contextual activities.

Existing applications implicitly assume that context collecting devices share the same notion of time. Thus, the concurrency property can be detected easily [3], [4], [5], [8]. However, this assumption does not necessarily hold in pervasive computing scenarios, which are characterized by the intrinsic asynchrony among computing entities [2], [9], [10], [11], [12], [13], [14], [15]. Specifically, context collecting devices may not have synchronized clocks and may run at different speeds. They heavily rely on wireless communications, which suffer from arbitrary delay [11]. Moreover, due to resource constraints, context collecting devices (usually resource-constrained sensors) often postpone the dissemination of context data, also resulting in asynchrony [10].

For example, in a smart office scenario, the mobile phone can provide useful contexts, such as "Bob is having a phone call" or "Bob's phone connects to the access point in the meeting room." However, the mobile phone is not

- *Y. Huang, Y. Yang, X. Ma, X. Tao, and J. Lu are with the Department of Computer Science and Technology, Nanjing University, 22# Hankou Road, Nanjing 210093, Jiangsu Province, China. E-mail: {csyuhuang, csylyang, xiaoxing.ma}@gmail.com, {txp, lj}@nju.edu.cn.*
- *J. Cao is with the Department of computing, Hong Kong Polytechnic University, Kowloon, Hong Kong, China. E-mail: csjcao@comp.polyu.edu.hk.*

Manuscript received 27 Dec. 2010; revised 4 May 2011; accepted 20 May 2011; published online 13 June 2011.
Recommended for acceptance by S. Gupta.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2010-12-0749.
Digital Object Identifier no. 10.1109/TPDS.2011.176.

necessarily synchronized with other context collecting devices, and Bob may not permit such synchronization. There are also cases where only partial order among contextual activities is sufficient [2]. Envision an elderly care scenario, where we only need to be aware that the elderly takes medicine before going to bed. In this case, we may not require periodic synchronization among all the context collecting devices involved.

We argue that in asynchronous pervasive computing scenarios, the concurrency property must be specified explicitly and detected at runtime. Toward this objective, we identify and address three essential issues:

- *Modeling of Environment Behavior*. We apply the classical logical time [16], [17] to model behavior of the asynchronous pervasive computing environment. One key notion in the modeling is the lattice structure among all meaningful observations of the asynchronous environment [18], [19].
- *Specification of the Concurrency Property*. We give the formal syntax for specification of the concurrency property. We also give the semantic interpretation based on the modeling of environment behavior.
- *Detection of the Concurrency Property.* We propose the *Concurrent contextual Activity Detection in Asynchronous environments* (CADA) algorithm, which achieves runtime detection of the concurrency property.

We further analyze the performance of CADA in pervasive computing scenarios. We also implement CADA over MIPA—the open-source context-aware middleware we developed [15], [20], and a case study is conducted. The performance analysis and measurements show that CADA effectively detects the concurrency property even when faced with dynamic changes in asynchrony of environment, duration of contextual activities, and the number of context collecting devices.

The rest of this paper is organized as follows: Sections 2, 3, and 4 discuss the essential issues of modeling, specification, and detection, respectively. Section 5 presents the case study and Section 6 outlines the related work. In Section 7, we conclude the paper with a brief summary and discuss the future work.

2 Modeling of Environment Behavior

The detection of contextual properties assumes the availability of an underlying context-aware middleware [3], [4], [15]. The middleware accepts the contextual property specified by the application, detects it at runtime and informs the application of the results (see more discussions in Section 5). Specifically, a collection of *nonchecker processes* $P^{(1)}, P^{(2)}, \ldots, P^{(n)}$ are deployed to monitor certain aspects of the environment. Examples of nonchecker processes are software processes manipulating networked physical sensors. One *checker process* P_{che} collects context data from nonchecker processes, and detects specified property at runtime. P_{che} is usually a third-party service deployed on the middleware. More notations as well as their explanations are listed in Table 1.

2.1 Message Passing and Logical Time

We model the nonchecker processes as a loosely-coupled message-passing system, without any global clock or shared memory. Communications suffer from finite but arbitrary delay. Dissemination of context data may be postponed due to resource constraints. We assume that no messages are lost, altered, or spuriously introduced, and use message sequence numbers to ensure that P_{che} receives messages from each $P^{(k)}$ in FIFO manner [21], [22], [23].

TABLE 1
Notations in Modeling of Environment Behavior

Notation	Explanation
$P^{(k)}$	nonchecker process ($1 \leqslant k \leqslant n$)
P_{che}	checker process
$s_i^{(k)}$	local state on $P^{(k)}$
$e_i^{(k)}$	local contextual event on $P^{(k)}$
$VC^{(k)}$	vector clock timestamp on $P^{(k)}$
$VC^{(k)}[i]$	i^{th} element of $VC^{(k)}$
$A^{(k)}$	contextual activity on $P^{(k)}$
A	Cartesian product of $A^{(1)}, A^{(2)}, \cdots, A^{(n)}$
$\mathcal{C}$	Consistent Global State (CGS)
$\mathcal{C}[k]$	k^{th} element of $\mathcal{C}$
$\mathcal{S}(\mathcal{C}_i, \mathcal{C}_j)$	CGS sequence from $\mathcal{C}_i$ to $\mathcal{C}_j$
$Lat(\mathcal{C}_0, \mathcal{C}_t)$	lattice of CGSs, starting at $\mathcal{C}_0$, ending at $\mathcal{C}_t$
$Bhv_{Lat(\mathcal{C}_0,\mathcal{C}_t)}$	possible CGS sequences of environment behavior

We reinterpret the notion of time based on the classical Lamport's definition of the happen-before (denoted by "$\rightarrow$") relation resulting from message passing [16], and its "on-the-fly" coding given by logical vector clocks [17]. Detailed definition of the happen-before relation and the vector clock can be found in Section 2 of the supplementary file, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.176.

2.2 Lattice of Consistent Global States

One key notion in modeling of environment behavior is the *Consistent Global State* (CGS) [18]. A *global state* is defined as a vector of local states from each $P^{(k)}$. If the constituent local states of a global state $\mathcal{C}$ are pairwise concurrent, $\mathcal{C}$ is a CGS, i.e.,

$$\mathcal{C} = \left(s^{(1)}, s^{(2)}, \ldots, s^{(n)}\right), \quad \forall\, i, j : 1 \leq i \neq j \leq n :: \neg\left(s^{(i)} \rightarrow s^{(j)}\right).$$

The CGS denotes a snapshot or meaningful observation of the asynchronous environment.

It is intuitive to define the *precede* (denoted by "$\prec$") relation between two CGSs: $\mathcal{C} \prec \mathcal{C}'$ if $\mathcal{C}'$ is obtained via advancing $\mathcal{C}$ by exactly one local state, i.e.,

$$\mathcal{C} \prec \mathcal{C}' \stackrel{def}{=\!=} \exists\ k, \text{ such that } \mathcal{C}'[k] \text{ is the first local state after } \mathcal{C}[k], \forall\ l : 1 \leq l \neq k \leq n :: \mathcal{C}[l] = \mathcal{C}'[l].$$

The *lead-to* relation (denoted by "$\rightsquigarrow$") is defined as the transitive closure of "$\prec$," i.e.,

$$\mathcal{C} \rightsquigarrow \mathcal{C}' \stackrel{def}{=\!=} \mathcal{C} \prec \mathcal{C}', or\ \exists\ \mathcal{C}_1, \mathcal{C}_2, \ldots \mathcal{C}_k, \quad \mathcal{C} \prec \mathcal{C}_1 \prec \cdots \prec \mathcal{C}_k \prec \mathcal{C}' (k = 1, 2, \cdots).$$

The set of all CGSs with the "$\rightsquigarrow$" relation define a lattice [18], [19], as shown in Fig. 1. This lattice structure serves as a key notion in the specification and detection of the concurrency property.

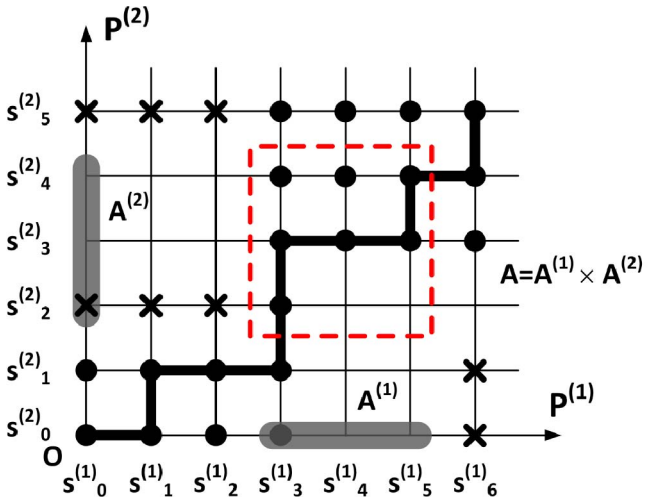


Fig. 1. The lattice structure among CGSs.

2.3 CGS Sequences in the Lattice

Behavior of the asynchronous pervasive computing environment can be viewed as the advancement through a sequence of CGSs connected by "$\prec$." Let $Lat(\mathcal{C}_0, \mathcal{C}_t)$ be the lattice with initial CGS $\mathcal{C}_0$ and latest CGS $\mathcal{C}_t$. We first define *CGS sequence* $\mathcal{S}(\mathcal{C}_i, \mathcal{C}_j)$ as

$$\mathcal{S}(\mathcal{C}_i, \mathcal{C}_j) = \mathcal{C}_i \mathcal{C}_{i+1} \cdots \mathcal{C}_j, 0 \le i \le j \le t, \\ \forall k : i \le k \le j-1 :: \mathcal{C}_k \prec \mathcal{C}_{k+1}.$$

Then, behavior of the asynchronous environment can be delineated by the potentially infinite CGS sequence $\mathcal{S}(\mathcal{C}_0, \mathcal{C}_i)$, which begins at $\mathcal{C}_0$ and currently ends at $\mathcal{C}_i$. Informally, $\mathcal{C}_i$ is one of the CGSs on the boundary of $Lat(\mathcal{C}_0, \mathcal{C}_t)$, i.e., its successors are yet to arrive in the future (see formal definition below). For example, $(s_3^{(1)}, s_5^{(2)})$ in Fig. 1 is such a "boundary" CGS.

Due to the intrinsic uncertainty in asynchronous environments, P_{che} obtains multiple possible CGS sequences of environment behavior. However, it does not know which one denotes the actual environment behavior [18], [19]. The set of all possible CGS sequences of environment behavior is defined as:

$$Bhv_{Lat(\mathcal{C}_0, \mathcal{C}_t)} = \{\mathcal{S}(\mathcal{C}_0, \mathcal{C}_i) | 0 \le i \le t, \\ \exists k, 1 \le k \le n, \mathcal{C}_i[k] = \mathcal{C}_t[k]\}.$$

For example, in Fig. 1, the possible CGS sequences of environment behavior begin at $\mathcal{C}_0 = (s_0^{(1)}, s_0^{(2)})$ and end at boundary CGSs $(s_3^{(1)}, s_5^{(2)})$, $(s_4^{(1)}, s_5^{(2)})$, $(s_5^{(1)}, s_5^{(2)})$, $(s_6^{(1)}, s_5^{(2)})$, $(s_6^{(1)}, s_4^{(2)})$, and $(s_6^{(1)}, s_3^{(2)})$.

The notion of $Bhv_{Lat(\mathcal{C}_0, \mathcal{C}_t)}$ serves as the basis for the semantic interpretation of the concurrency property in Section 3.2.

3 Specification of the Concurrency Property

In this section, we present the logic for specification of the concurrency property. We first give the syntax and then discuss the semantic interpretation.

3.1 Syntax

Syntax for the concurrency property CP is as follows, and we explain it in a bottom-up manner

$$CP ::= Def(\phi), \\ \phi ::= \phi^{(1)} \wedge \phi^{(2)} \wedge \cdots \wedge \phi^{(n)}, \\ \phi^{(k)} ::= \text{local predicate on } P^{(k)} (1 \le k \le n).$$

Local predicate $\phi^{(k)}$ is specified over local states on $P^{(k)}$ to delineate the application's concern about specific aspect of the pervasive computing environment. Though $\phi^{(k)}$ may be complex, its value only depends on local information with respect to $P^{(k)}$. Since we focus on asynchrony of a distributed environment, the local predicate is considered atomic and its detection scheme is assumed available.

Informally, concurrency means that on some time instant, every contextual activity is taking place. However in asynchronous environments, nonchecker processes do not share the same notion of time. The notion of the "same time instant" should be interpreted as a global state whose constituent local states could possibly take place on the same time instant (do not have explicit happen-before relation), i.e., a CGS.

Thus, predicate ϕ is specified over some CGS $\mathcal{C}_\phi$ to describe the concurrency among contextual activities. It is defined as the conjunction of local predicates. Every constituent local predicate $\phi^{(k)}$ is specified over local state $\mathcal{C}_\phi[k]$.

As discussed in Section 2.3, P_{che} only knows that the environment behavior can be delineated by one of the multiple possible CGS sequences. Thus, specification of CGS predicate ϕ does not make sense in asynchronous environments. Modal operator $Def(\cdot)$ is introduced to make ϕ meaningful over multiple possible CGS sequences (see detailed semantic interpretation below).

3.2 Semantic Interpretation

As discussed in Section 3.1, $\phi^{(k)}$ is considered atomic, and the scheme for detecting the satisfaction of $\phi^{(k)}$, i.e., $s^{(k)} \models \phi^{(k)}$, is assumed available. Upon each update of the context data, $P^{(k)}$ updates the value of $\phi^{(k)}$ accordingly.

As for CGS predicate ϕ, if every constituent local predicate $\phi^{(k)}$ is true on the constituent local state $\mathcal{C}[k]$, we say that satisfaction of ϕ is detected over $\mathcal{C}$, i.e.,

$$\mathcal{C} \models \phi \overset{def}{=\!=} \forall\ k : 1 \le k \le n :: \mathcal{C}[k] \models \phi^{(k)}.$$

If ϕ holds on some CGS $\mathcal{C}_k$ within CGS sequence $\mathcal{S}(\mathcal{C}_i, \mathcal{C}_j)$, we say that satisfaction of ϕ is detected over CGS sequence $\mathcal{S}(\mathcal{C}_i, \mathcal{C}_j)$, i.e.,

$$\mathcal{S}(\mathcal{C}_i, \mathcal{C}_j) \models \phi \overset{def}{=\!=} \exists\ \mathcal{C}_k \in \mathcal{S}(\mathcal{C}_i, \mathcal{C}_j), \mathcal{C}_k \models \phi.$$

Informally, modal operator $Def(\phi)$ means that ϕ holds on every possible CGS sequences, i.e.,

$$Lat(\mathcal{C}_0, \mathcal{C}_t) \models Def(\phi) \overset{def}{=\!=} \forall\ \mathcal{S} \in Bhv_{Lat(\mathcal{C}_0, \mathcal{C}_t)}, \mathcal{S} \models \phi.$$

Since the context data are intrinsically error-prone in pervasive computing scenarios, we adopt $Def(\cdot)$ to gain more confidence on that the concurrency property of interest actually holds.

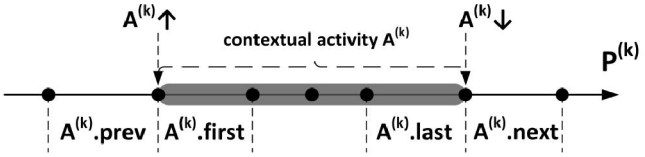


Fig. 2. States and events concerning an activity.

4 Detection of the Concurrency Property

In this section, we propose the Concurrent contextual Activity Detection in Asynchronous environments algorithm to achieve runtime detection of the concurrency property. We explain the main design rationale of CADA with a concrete example. Detailed design and performance analysis of CADA in pervasive computing scenarios can be found in Sections 6 and 7 of the supplementary file, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.176.

4.1 Satisfaction of the Concurrency Property and Overlapping among Contextual Activities

The correctness of CADA is decided by whether it reports satisfaction of the concurrency property conforming to the semantic interpretation. We first show the equivalence between *satisfaction of the concurrency property* (formally defined in Section 3.2) and *the overlapping among contextual activities* (formally defined in (1) below). Then, we outline the design of CADA, which detects the concurrency property by detection of overlapping among contextual activities.

By *contextual activity* $A^{(k)}$, we refer to the longest period (consisting of consecutive local states) on $P^{(k)}$, in which local predicate $\phi^{(k)}$ holds, i.e.,

$$A^{(k)} \stackrel{def}{=\!=} \left[s_r^{(k)}, s_{r+1}^{(k)}, \ldots, s_{r+l}^{(k)}\right], \forall\, s \in A^{(k)}, s \models \phi^{(k)}.$$

To facilitate discussions below, we define four local states and two contextual events concerning $A^{(k)}$, as shown in Fig. 2. $A^{(k)}.first$ defines the first local state in $A^{(k)}$. $A^{(k)}.prev$ denotes the last local state before $A^{(k)}$. The contextual event connecting $A^{(k)}.prev$ and $A^{(k)}.first$ is denoted as $A^{(k)}\uparrow$. Notations $A^{(k)}.next$, $A^{(k)}.last$ and $A^{(k)}\downarrow$ are duals of $A^{(k)}.prev$, $A^{(k)}.first$ and $A^{(k)}\uparrow$, respectively.

The overlapping among contextual activities is defined as

$$\forall\, i, j : 1 \le i \ne j \le n :: A^{(i)}\uparrow \rightarrow A^{(j)}\downarrow. \tag{1}$$

As an example, the overlapping between two activities is shown in Fig. 3.

The correctness of CADA depends on Theorem 1 below:

Theorem 1. *$Def(\phi)$ is true, if and only if the contextual activities involved in ϕ satisfy (1).*

We prove Theorem 1 by the following Lemmas 2 and 3. Sections 4.1.1 and 4.1.2 illustrate rationale of the proof via the example in Fig. 1.

4.1.1 Overlapping Implies $Def(\phi)$

Given the overlapping among contextual activities, in order to prove the satisfaction of $Def(\phi)$, we need to prove that, $\forall\, \mathcal{S}(\mathcal{C}_0, \mathcal{C}_i) \in Bhv_{Lat(\mathcal{C}_0, \mathcal{C}_t)}, \mathcal{S}(\mathcal{C}_0, \mathcal{C}_i) \models \phi$.

If ϕ holds on some CGS $\mathcal{C}$, each of the constituent $\phi^{(k)}$ is true on $\mathcal{C}[k]$, i.e., $\mathcal{C}[k]$ is within some contextual activity $A^{(k)}$. Thus,

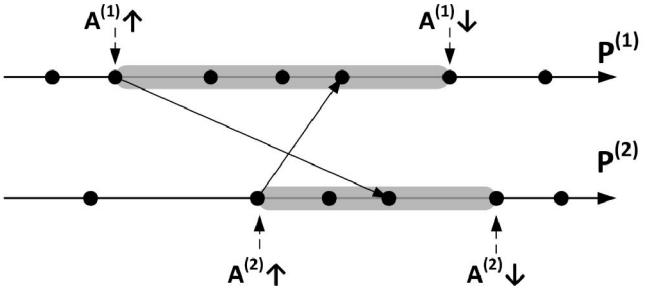


Fig. 3. Concurrency between two intervals.

$\mathcal{C} \models \phi$ means that $\mathcal{C}$ is inside the n-dimensional rectangle $A = A^{(1)} \times A^{(2)} \times \cdots \times A^{(n)}$. The geometric interpretation is that each of the possible CGS sequences of environment behavior will go across A. As for the example in Fig. 1, $A = A^{(1)} \times A^{(2)}$, as indicated by the dotted-line rectangle.

In our example, we instantiate (1) as: $s_2^{(1)} \rightarrow s_5^{(2)} \wedge s_1^{(2)} \rightarrow s_6^{(1)}$. Since $\forall\, k < 2, s_k^{(1)} \rightarrow s_2^{(1)} \rightarrow s_5^{(2)}$, we have that

$$\forall\, 0 \le k \le 2, \left(s_k^{(1)}, s_5^{(2)}\right) \text{ is not a CGS.}$$

This means that none of the global states in the horizontal line segment between global states $(s_0^{(1)}, s_5^{(2)})$ and $(s_2^{(1)}, s_5^{(2)})$ is a CGS, as denoted by the upper horizontal line of crosses in Fig. 1.

Similarly, since $s_1^{(2)} \rightarrow s_6^{(1)}$, we have that: $\forall\, 0 \le k \le 1$, $(s_6^{(1)}, s_k^{(2)})$ is not a CGS. This means that none of the global states in the vertical line segment between global states $(s_6^{(1)}, s_0^{(2)})$ and $(s_6^{(1)}, s_1^{(2)})$ is a CGS, as denoted by the vertical line of crosses in Fig. 1.

Since behavior of the environment can only proceed through the CGS sequence, we have that the computation can never go across the lines of inconsistent global states. Thus, as shown in Fig. 1, the CGS sequence must pass the rectangle $A = A^{(1)} \times A^{(2)}$, as forced by the "fences" formed of inconsistent global states. According to the discussions above, we derive the following lemma:

Lemma 2. *The overlapping among contextual activities (as defined by (1)) implies $Def(\phi)$.*

Proof. See Section 3 of the supplementary file, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.176. □

4.1.2 $Def(\phi)$ Implies Overlapping

Given that $Def(\phi)$ holds, we need to prove that (1) holds. We show this by contradiction. If (1) does not hold, we have that: $\exists i, j, \neg(A^{(i)}.prev \rightarrow A^{(j)}.next)$. In our example, this is instantiated as

$$\neg\left(s_2^{(1)} \rightarrow s_5^{(2)}\right) \text{ or } \neg\left(s_1^{(2)} \rightarrow s_6^{(1)}\right).$$

We focus on the first case, and arguments for the latter case are similar. Given that $\neg(s_2^{(1)} \rightarrow s_5^{(2)})$, we also need to consider two cases:

- If $s_5^{(2)} \rightarrow s_2^{(1)}$, we have that $A^{(2)}\downarrow \rightarrow A^{(1)}\uparrow$. Thus, $A^{(1)}$ and $A^{(2)}$ do not overlap, which contradicts with that $Def(\phi)$ holds.
- If $\neg(s_5^{(2)} \rightarrow s_2^{(1)})$, we have that $(s_2^{(1)}, s_5^{(2)})$ is a CGS. Thus, the CGS sequence of environment behavior

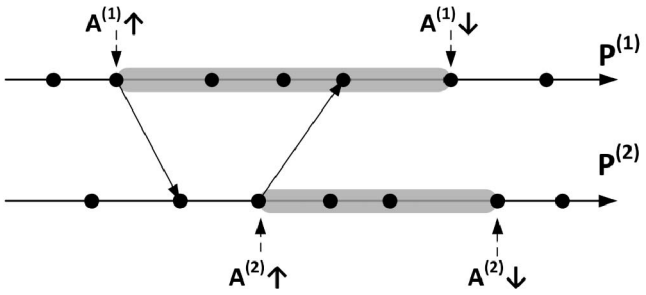


Fig. 4. Early detection.

can possibly pass $(s_2^{(1)}, s_5^{(2)})$. Once the CGS sequence arrives at $(s_2^{(1)}, s_5^{(2)})$, it will circumvent A. This also contradicts with that $Def(\phi)$ holds.

Discussions above indicate the following lemma:

Lemma 3. *$Def(\phi)$ implies overlapping among contextual activities (as defined by (1)).*

Proof. See Section 4 of the supplementary file, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.176. □

4.2 Early Detection

From Fig. 1, we can see that if $s_2^{(1)} \rightarrow s_2^{(2)}$, we have that none of the global states between $(s_2^{(1)}, s_2^{(2)})$ and $(s_0^{(1)}, s_2^{(2)})$ is a CGS, as shown by the lower horizontal line segment of crosses in Fig. 1. In this case, the CGS sequences of environment behavior are restricted to a narrower region, and still must go across A. Thus, $Def(\phi)$ still holds. This more stringent requirement corresponds to the following formula:

$$A^{(1)} \uparrow \rightarrow A^{(2)} \uparrow \wedge A^{(2)} \uparrow \rightarrow A^{(1)} \downarrow . \quad (2)$$

As shown in Fig. 4, if (2) holds, the overlapping can be decided with only three timestamps of $A^{(1)} \uparrow$, $A^{(2)} \uparrow$, and $A^{(1)} \downarrow$. Thus, CADA has chances to detect the overlapping earlier. We name this case *early detection*.

Equation (2) can be generalized to

$$\exists\, i :: \big(\forall j : j \neq i :: A^{(j)} \uparrow \rightarrow A^{(i)} \uparrow \wedge A^{(i)} \uparrow \rightarrow A^{(j)} \downarrow\big). \quad (3)$$

Based on the discussions above, we have the following corollary:

Corollary 4. *If there exists one contextual activity satisfying (3), while all other activities satisfy (1), $Def(\phi)$ holds.*

Proof. See Section 5 of the supplementary file, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.176. □

4.3 Design and Analysis of the CADA Algorithm

The CADA algorithm runs on both the nonchecker process side and the checker process side:

- On the nonchecker process side, CADA detects satisfaction of the local predicate. It also let the nonchecker processes send messages among each other, in order to build the happen-before relation required by (1) and (3). Vector clock timestamps of local states are sent to the checker process.
- On the checker process side, the collected logical timestamps are compared to detect whether the specified concurrency property holds, and whether early detection could be achieved.

Detailed design and pseudocodes of the CADA algorithm are presented in Section 6 of the supplementary file, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.176.

Given design of the proposed CADA algorithm, we also need to explore that whether CADA works well in pervasive computing environments. Specifically, can CADA achieve accurate detection of the concurrency property? Can CADA detect the concurrency property in time? We address these issues by theoretical performance analysis in pervasive computing scenarios. Quantitative analytical results concerning the accuracy and timeliness of CADA are presented. We also further illustrate our analysis by numerical results. Please refer to Section 7 of the supplementary file, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.176, for details.

5 Case Study

In this section, we further investigate the performance of CADA via a case study. We first demonstrate how to apply the CADA algorithm in a smart-office scenario. Then, we outline results concerning the performance measurements of CADA in this scenario.

5.1 The Smart-Office Scenario

We investigate a smart office scenario, where a context-aware application on Bob's mobile phone automatically turns the phone to silent mode when Bob attends a lecture. In our scenario, we assume that sensors can detect physical events with sufficient accuracy. For example, the sensor can accurately detect "Bob enters room A" and "Bob leaves room A". Thus, we can obtain the contextual activity that "Bob is in room A" (the issue of coping with inaccurate or erroneous sensor readings is not covered in this work).

The location context is detected by Bob's mobile phone. When the phone connects to the access point in the meeting room, we assume that Bob is in this room. A nonchecker process $P^{(1)}$ is deployed over Bob's mobile phone, which periodically updates the phone's connection to access points. We detect that a presentation is going on if the projector is working, and nonchecker process $P^{(2)}$ is deployed over the projector.

5.2 Specification and Detection of the Concurrency Property

Observe that the mobile phone and the projector do not necessarily have synchronized clocks, and Bob may not be willing to synchronize his mobile with other devices due to security concerns. Moreover, the smart phone may also adaptively postpone the update of location context to save battery power.

Given the assumption that physical activity can be accurately sensed, the nonchecker processes can then send control messages right after the beginning of contextual

activities. We have shown in Section 4.1 that using the happen-before relation built by control messages, we can detect the overlapping among contextual activities, thus detecting specified concurrency property.

In our scenario, the application detects that Bob attends a lecture by specification of the concurrency property: C_1: *location of Bob is the meeting room and a presentation is going on in the room* (first discussed in Section 1). Formally,

$$C_1 = Def(\phi_1 \wedge \phi_2),$$

where local predicates

$$\begin{aligned}\phi_1 &= \text{the user's smart phone connects to the} \\ &\quad \text{access point inside the meeting room} \\ \phi_2 &= \text{the projector is working.}\end{aligned}$$

5.3 Performance Measurements

The CADA algorithm is implemented over the open-source context-aware middleware MIPA we developed [15], [20], and the smart-office scenario is simulated. Please refer to Section 8 of the supplementary file, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.176, for detailed discussions on the experiment configurations and the experimental evaluation results.

According to the performance evaluation results, we find that CADA is desirable for context-aware applications in asynchronous pervasive computing scenarios. In particular: 1) CADA can tolerate a reasonably large amount of asynchrony (in terms of message delay and interval between context updates), and the interval between context updates has more impact than the message delay; 2) the duration of contextual activities does not affect the detection of overlapping among activities, but does impact the odds of early detection; 3) the number of activities affects the detection of overlapping among contextual activities, as well as the odds of early detection.

6 Related Work

Our work can be positioned against two areas of existing work: context-aware computing and detection of global predicates in distributed computations.

As for context-aware computing, various schemes have been proposed for detection of contextual properties in the literature. However, 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 environments.

The problem of detecting global predicates mainly arises from the debugging of distributed programs. Existing predicate detection schemes are mainly nonintrusive, and passively piggyback over the programs under debugging. In comparison, CADA is intrusive, i.e., it makes computing entities proactively send messages to each other, to build the happen-before relation it requires. We argue that this is necessary to achieve context-awareness in asynchronous environments.

A more detailed discussion on the related work is presented in Section 9 of the supplementary file, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.176.

7 Conclusion

In this work, we study how to detect the concurrency property in asynchronous pervasive computing environments. Toward this objective, our contributions are: 1) we apply the classical logical time to cope with the asynchrony in pervasive computing scenarios; 2) we present the specification of the concurrency property in asynchronous environments, and propose the CADA algorithm to detect specified concurrency property at runtime; 3) we evaluate CADA by both theoretical analysis and experiments.

Currently, the CADA algorithm still suffers from several limitations. In our future work, we need to study how to detect properties concerning dynamic behavior of the environment. We need to investigate how to explicitly and effectively maintain the lattice of CGSs at runtime and detect different types of contextual properties based on the lattice. It is also important to investigate how to reduce the message complexity of our proposed property detection schemes. Moreover, we will investigate the application of predicate detection schemes in other scenarios, e.g., the Internet-of-Things.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (No. 60903024, 60736015, 61021062), the National 973 Program of China (2009CB320702), and the "Climbing" Program of Jiangsu Province, China (BK2008017).

References

[1] D. Garlan, D. Siewiorek, A. Smailagic, and P. Steenkiste, "Project Aura: Toward Distraction-Free Pervasive Computing," *IEEE Pervasive Computing*, vol. 1, no. 2, pp. 22-31, Apr.-June 2002.

[2] A. Dey, "Providing Architectural Support for Building Context-Aware Applications," PhD thesis, Georgia Inst. of Technology, Nov. 2000.

[3] C. Xu and S.C. Cheung, "Inconsistency Detection and Resolution for Context-Aware Middleware Support," *Proc. ACM SIGSOFT Int'l Symp. Foundations of Software Eng. (FSE '05)*, pp. 336-345, Sept. 2005.

[4] C. Xu, S.C. Cheung, W.K. Chan, and C. Ye, "Partial Constraint Checking for Context Consistency in Pervasive Computing," *ACM Trans. Software Eng. and Methodology*, vol. 19, no. 3, pp. 1-61, 2010.

[5] Y. Huang, X. Ma, X. Tao, J. Cao, and J. Lu, "A Probabilistic Approach to Consistency Checking for Pervasive Context," *Proc. IEEE/IFIP Int'l Conf. Embedded and Ubiquitous Computing (EUC '08)*, pp. 387-393, Dec. 2008.

[6] S.R. Jeffery, M. Garofalakis, and M.J. Franklin, "Adaptive Cleaning for Rfid Data Streams," *Proc. Int'l Conf. Very Large Data Bases (VLDB '06)*, pp. 163-174, Sept. 2006.

[7] A. Ranganathan, R.H. Campbell, A. Ravi, and A. Mahajan, "Conchat: A Context-Aware Chat Program," *IEEE Pervasive Computing*, vol. 1, no. 3, pp. 51-57, http://dx.doi.org/10.1109/MPRV.2002.1037722, July 2002.

[8] Y. Huang, J. Cao, B. Jin, X. Tao, J. Lu, and Y. Feng, "Flexible Cache Consistency Maintenance over Wireless Ad Hoc Networks," *IEEE Trans. Parallel and Distributed Systems*, vol. 21, no. 8, pp. 1150-1161, Aug. 2010.

[9] E.A. Lee, "Cyber-Physical Systems—Are Computing Foundations Adequate?," *Proc. US Nat'l Science Foundation (NSF) Workshop Cyber-Physical Systems: Research Motivation, Techniques and Roadmap, Position Paper*, 2006.

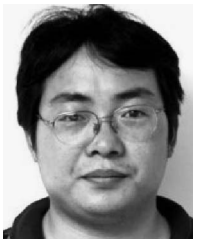
[10] M. Sama, D.S. Rosenblum, Z. Wang, and S. Elbaum, "Model-Based Fault Detection in Context-Aware Adaptive Applications," *Proc. 16th ACM SIGSOFT Int'l Symp. Foundations of Software Eng. (SIGSOFT '08/FSE-16).* pp. 261-271, 2008.
[11] Y. Huang, X. Ma, J. Cao, X. Tao, and J. Lu, "Concurrent Event Detection for Asynchronous Consistency Checking of Pervasive Context," *Proc. IEEE Int'l Conf. Pervasive Computing and Comm. (PERCOM '09),* Mar. 2009.
[12] L. Kaveti, S. Pulluri, and G. Singh, "Event Ordering in Pervasive Sensor Networks," *Proc. IEEE Int'l Conf. Pervasive Computing and Comm. Workshops (PERCOMW '09),* pp. 604-609, Mar. 2009.
[13] Y. Huang, J. Yu, J. Cao, and X. Tao, "Detection of Behavioral Contextual Properties in Asynchronous Pervasive Computing Environments," *Proc. Int'l Conf. Parallel and Distributed Systems (ICPADS '10),* Dec. 2010.
[14] T. Hua, Y. Huang, J. Cao, and X. Tao, "A Lattice-Theoretic Approach to Runtime Property Detection for Pervasive Context," *Proc. Int'l Conf. Ubiquitous Intelligence and Computing (UIC '10),* pp. 307-321, Oct. 2010.
[15] J. Yu, Y. Huang, J. Cao, and X. Tao, "Middleware Support for Context-Awareness in Asynchronous Pervasive Computing Environments," *Proc. Int'l Conf. Embedded and Ubiquitous Computing (EUC '10),* Dec. 2010.
[16] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM,* vol. 21, no. 7, pp. 558-565, 1978.
[17] F. Mattern, "Virtual Time and Global States of Distributed Systems," *Proc. Int'l Workshop Parallel and Distributed Algorithms,* pp. 215-226, 1989.
[18] O. Babaoğlu and K. Marzullo, "Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms," *Distributed Systems,* second ed., Chapter 4, pp. 55-96, ACM Press, 1993.
[19] R. Schwarz and F. Mattern, "Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail," *J. Distributed Computing* , vol. 7, no. 3, pp. 149-174, 1994.
[20] "MIPA—Middleware Infrastructure for Predicate Detection in Asynchronous Environments," http://mipa.googlecode.com, 2011.
[21] V. Garg and B. Waldecker, "Detection of Weak Unstable Predicates in Distributed Programs," *IEEE Trans. Parallel and Distributed Systems,* vol. 5, no. 3, pp. 299-307, Mar. 1994.
[22] V.K. Garg and B. Waldecker, "Detection of Strong Unstable Predicates in Distributed Programs," *IEEE Trans. Parallel and Distributed Systems,* vol. 7, no. 12, pp. 1323-1333, Dec. 1996.
[23] P. Chandra and A.D. Kshemkalyani, "Causality-Based Predicate Detection across Space and Time," *IEEE Trans. Computers,* vol. 54, no. 11, pp. 1438-1453, Nov. 2005.

Yu Huang received the BSc and PhD degrees in computer science from the University of Science and Technology of China, Hefei, China, in 2002 and 2007, respectively. From 2003 to 2007, he studied in the Institute of Software, Chinese Academy of Sciences, as a coeducated PhD student. He also studied in the Department of Computing, Hong Kong Polytechnic University, as an exchange student from September 2005 to September 2006. He is currently an associate professor in the Department of Computer Science and Technology, Nanjing University, China. His research interests include distributed computing theory, formal specification and verification, and pervasive context-aware computing. He is a member of the IEEE and the China Computer Federation.

Yiling Yang received the BSc degree in computer science from Nanjing University, China, in 2009, where he is now working toward the PhD degree in computer science. His research interests include software engineering and methodology, theory of distributed computing, middleware technologies and pervasive computing.

Jiannong Cao received the BSc degree in computer science from Nanjing University, Nanjing, China, in 1982, and the MSc and PhD degrees in computer science from Washington State University, Pullman, in 1986 and 1990, respectively. He is currently a professor in the Department of Computing at Hong Kong Polytechnic University, Hung Hom, Hong Kong. He is also the director of the Internet and Mobile Computing Lab in the department. Before joining Hong Kong Polytechnic University, he was on the faculty of computer science at James Cook University and University of Adelaide in Australia, and City University of Hong Kong. His research interests include parallel and distributed computing, networking, mobile and wireless computing, fault tolerance, and distributed software architecture. He has published more than 200 technical papers in the above areas. His recent research has focused on mobile and pervasive computing systems, developing test-bed, protocols, middleware, and applications. He has served as a member of editorial boards of several international journals, a reviewer for international journals/conference proceedings, and also as an organizing/program committee member for many international conferences. He is a senior member of China Computer Federation, a senior member of the IEEE, including Computer Society and the IEEE Communication Society, and a member of the ACM. He is also a member of the IEEE Technical Committee on Distributed Processing, IEEE Technical Committee on Parallel Processing, IEEE Technical Committee on Fault Tolerant Computing.

Xiaoxing Ma received the BSc and PhD degrees in Nanjing University, China, both in computer science. He is currently a professor in the Department of Computer Science and Technology at Nanjing University. His research interests include software methodology and software adaptation. He is a member of the IEEE.



Xianping Tao received the BSc degree from National University of Defense Technology, Changsha, China, and the MSc and PhD degrees in Nanjing University, Nanjing China, all in computer science. He is currently a professor in the Department of Computer Science and Technology at Nanjing University. His research interests include software engineering and methodology, middleware systems, and pervasive computing. He is a member of the IEEE.

Jian Lu received the BSc, MSc and PhD degrees in computer science from Nanjing University, P.R. China. He is currently a professor in the Department of Computer Science and Technology and the director of the State Key Laboratory for Novel Software Technology at Nanjing University. He serves on the Board of the International Institute for Software Technology of the United Nations University (UNU-IIST). He also serves as the director of the Software Engineering Technical Committee of the China Computer Federation. His research interests include software methodologies, software automation, software agents, and middleware systems.

Supplementary file of "Runtime Detection of the Concurrency Property in Asynchronous Pervasive Computing Environments"

Yu Huang,Yiling Yang, Jiannong Cao,Xiaoxing Ma,Xianping Tao,and Jian Lu

1 Introduction

In our paper "Runtime Detection of the Concurrency Property in Asynchronous Pervasive Computing Environments", we have discussed how to achieve context-awareness in asynchronous pervasive computing environments. Three essential issues are identified and addressed: modeling of environment behavior, specification of the concurrency property, and detection of the concurrency property by the CADA algorithm.

In this supplementary file, we further provide more details about how we address these three essential issues. Specifically, Section 2 gives definition of the happen-before relation and logical vector time. Section 3, 4 and 5 provide proofs of Lemma 2, Lemma 3 and Corollary 4 respectively. Section 6 and 7 provide detailed design and performance analysis of CADA respectively. Section 8 discusses the performance measurements and Section 9 provides additional discussions on the related work.

Please note that, unless explicitly stated, all references to sections, formulas, figures, tables, algorithms and references are referring to entities in this supplementary file.

2 Definition of the Happen-before Relation and Logical Vector Time

We re-interpret the notion of time based on Lamport's definition of the *happen-before* relation (denoted by '$\rightarrow$') resulting from message causality [1], as well as its on-the-fly coding by logical vector clocks [2].

- *Yu Huang, Yiling Yang, Xiaoxing Ma, Xianping Tao and Jian Lu are with the State Key Laboratory for Novel Software Technology, and Department of Computer Science and Technology, Nanjing University, Nanjing, China, 210093.*
  E-mail: {yuhuang, xxm, txp, lj}@nju.edu.cn, csylyang@gmail.com
- *Jiannong Cao is with the Hong Kong Polytechnic University.*
  E-mail: csjcao@comp.polyu.edu.hk

Specifically, each non-checker process $P^{(k)}$ generates its (potentially infinite) trace of *local states* connected by *contextual events*: "$s_0^{(k)}, e_0^{(k)}, s_1^{(k)}, e_1^{(k)}, s_2^{(k)}, e_2^{(k)} \ldots$". The contextual event may be local, e.g. update of the context data, or global, e.g. sending/receiving messages.

For two local contextual events e_1 and e_2, we have $e_1 \rightarrow e_2$ iff:

- Events e_1 and e_2 are on the same non-checker process and e_1 is generated before e_2, or
- Events e_1 and e_2 are on different non-checker processes, and e_1 and e_2 are the corresponding sending and receiving of the same message respectively, or
- There exists some e_3 such that $e_1 \rightarrow e_3 \rightarrow e_2$.

For two local states, $s_1 \rightarrow s_2$ iff the ending of s_1 happen-before the beginning of s_2 (note that the beginning and ending of a state are both contextual events).

We use the logical vector clock to depict the happen-before relation. Specifically, each $P^{(j)}$ keeps its vector clock timestamp $VC^{(j)}$:

- $VC^{(j)}[i](i \neq j)$ is ID of the last message from $P^{(i)}$, which has a causal relation to $P^{(j)}$.
- $VC^{(j)}[j]$ is the next message ID $P^{(j)}$ will use.

3 Proof of Lemma 2

We prove Lemma 2 by contradiction. Assume that $Def(\phi) = Def(\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n)$ does not hold. Initially, state of the environment is at CGS $\mathcal{C}_0$ before the n-dimensional rectangle $A = A^{(1)} \times A^{(2)} \times \cdots \times A^{(n)}$, i.e.,

$$\forall\ i, \mathcal{C}_0[i] \rightarrow A^{(i)}.first$$

The computation will eventually reach some CGS $\mathcal{C}_{evt}$ after A, i.e.,

$$\forall\ i, A^{(i)}.last \rightarrow \mathcal{C}_{evt}[i]$$

Since $Def(\phi)$ does not hold, we have that,

$$\exists\ \mathcal{S} = \mathcal{S}(C_0, C_{evt}) : \forall\ \mathcal{C} \in \mathcal{S},\ \mathcal{C} \notin A$$

As the computation proceeds over $\mathcal{S}$, assume that the computation has passed through $A^{(i)}$, and is still before $A^{(j)}$, i.e. the computation has arrived at some CGS $\mathcal{C}$ such that

$$\begin{aligned} \mathcal{C}[j] = A^{(j)}.prev \ or\ \mathcal{C}[j] \rightarrow A^{(j)}.prev \\ A^{(i)}.next = \mathcal{C}[i] \ or\ A^{(i)}.next \rightarrow \mathcal{C}[i] \end{aligned} \quad (1)$$

According to Formula (2),

$$\forall\ i, j : 1 \leqslant i \neq j \leqslant n :: A^{(i)} \uparrow\!\!\rightarrow A^{(j)} \downarrow \quad (2)$$

we have that:

$$A^{(j)}.prev \rightarrow A^{(i)}.next \quad (3)$$

According to Formula(1) and (3), we have that $\mathcal{C}[j] \rightarrow \mathcal{C}[i]$. This contradicts with the fact that $\mathcal{C}$ is a CGS.

Thus, if there exists some non-checker process $P^{(i)}$ such that $P^{(i)}$ has not entered $A^{(i)}$ ($\mathcal{C}[i] = A^{(i)}.prev$ or $\mathcal{C}[i] \rightarrow A^{(i)}.prev$), none of other non-checker processes $P^{(j)}$ can go beyond $A^{(j)}$. This contradicts with that the computation does not intersect with A and eventually arrives at $\mathcal{C}_{evt}$.

Thus we prove that if the contextual activities satisfy Formula (2), $Def(\phi)$ holds. Note that the discussion above is the formal description of the fact that the computation must go across the n-dimensional rectangle $A = A^{(1)} \times A^{(2)} \times \cdots \times A^{(n)}$ as forced by the horizontal/vertical "fences" formed by inconsistent global states.

4 Proof of Lemma 3

We also prove Lemma 3 by contradiction. Assume that Formula (2) does not hold, i.e.,

$$\exists p, q, \neg(A^{(p)}.prev \rightarrow A^{(q)}.next)$$

which is equivalent to two cases: i) $A^{(q)}.next \rightarrow A^{(p)}.prev$, or ii) $A^{(p)}.prev || A^{(q)}.next$ ($s||s'$ is defined as $\neg(s \rightarrow s') \wedge \neg(s' \rightarrow s)$).

Case i) means that the ending of $A^{(q)}$ happens before the beginning of $A^{(p)}$. Obviously this contradicts with the fact that $Def(\phi)$ holds. So we focus on Case ii) below. We need to prove that there exists some CGS $\mathcal{C}$ such that

$$\mathcal{C}[p] = A^{(p)}.prev, \mathcal{C}[q] = A^{(q)}.next$$

i.e., for any other non-checker process $P^{(i)} (1 \leqslant i \leqslant n, i \neq p, i \neq q)$, we need to find local state s on $P^{(i)}$, such that $(s||A^{(p)}.prev) \wedge (s||A^{(q)}.next)$.

For the initial local state $s_0^{(i)}$, we have that $\neg(A^{(p)}.prev \rightarrow s_0^{(i)})$ because according to our system model, nothing happens before the initial state. Furthermore, if $\neg(s_0^{(i)} \rightarrow A^{(p)}.prev)$, we have already found $s_0^{(i)}$ such that $s_0^{(i)} || A^{(p)}.prev$. Otherwise if $s_0^{(i)} \rightarrow A^{(p)}.prev$, we consider the next local state $s_1^{(i)}$. Apply the arguments for $s_0^{(i)}$ again, we have that $s_1^{(i)} \rightarrow A^{(p)}.prev$ must hold.

According to the discussions above, we have that if none of local states on $P^{(i)}$ is concurrent with $A^{(p)}.prev$, there are infinite local states which happen-before $A^{(p)}.prev$. This contradicts with the axiom of finite causes [3]. Thus we prove that there must exist some local state on $P^{(i)}$ which is concurrent with $A^{(p)}.prev$. Similarly, we can prove that there must exist some local state on $P^{(i)}$ which is concurrent with $A^{(q)}.next$. Thus, we prove that there exists CGS $\mathcal{C}$ such that:

$$\mathcal{C}[p] = A^{(p)}.prev, \mathcal{C}[q] = A^{(q)}.next$$

For the computation which passes $\mathcal{C}$, since $\mathcal{C}[p] = A^{(p)}.prev$, we have that the computation has not entered A yet. Meanwhile, since $\mathcal{C}[q] = A^{(q)}.next$, we have that the computation will never pass A in the future. This leads to contradiction with that $Def(\phi)$ holds.

Thus we prove that if $Def(\phi)$ holds, the contextual activities satisfy Formula (2).

5 Proof of Corollary 4

If Formula (4) holds between some activity $A^{(i)}$ and every other activity $A^{(j)}$, we have that Formula (2) must also hold between $A^{(i)}$ and $A^{(j)}$.

$$\exists\ i :: (\forall j : j \neq i :: A^{(j)} \uparrow\!\!\rightarrow A^{(i)} \uparrow \wedge\ A^{(i)} \uparrow\!\!\rightarrow A^{(j)} \downarrow) \quad (4)$$

It is because $A^{(j)} \uparrow\!\!\rightarrow A^{(i)} \uparrow$ (by Formula (4)) and $A^{(i)} \uparrow\!\!\rightarrow A^{(i)} \downarrow$ (since the beginning of some activity must happen before its ending). Thus, $A^{(j)} \uparrow\!\!\rightarrow A^{(i)} \downarrow$ (by transitivity of the '$\rightarrow$' relation), as required by Formula (2).

Thus Formula (2) holds for all the activities and $Def(\phi)$ holds.

6 Detailed Design of the CADA Algorithm

The CADA algorithm runs on both the non-checker process side and the checker process side. Its operation involves two different types of messages:

- *Control message*. Non-checker processes send control messages among each other to build the happen-before relation required by Formula (2) and (4).
- *Checking message*. The checker process collects logical timestamps in checking messages from non-checker processes and detects the specified concurrency property.

The notations used in the design of CADA are listed in Table 1.

TABLE 1
Notations used in the design of CADA

Notation	Explanation
$lo^{(k)}/hi^{(k)}$	logical timestamps for $A^{(k)}\uparrow/A^{(k)}\downarrow$
$flag\_msg\_act$	boolean value for reducing redundant checking messages (initially $false$)
$MsgQue^{(k)}$	queue on $P^{(k)}$ storing messages to be sent (initially empty)
$Que^{(k)}$	queue on P_{che} storing time stamps from $P^{(k)}$ (initially empty)

6.1 CADA on the Non-checker Process Side

Each $P^{(k)}$ is in charge of updating local states and local predicates. It also proactively sends control messages upon the beginning of contextual activities. The happen-before relation resulting from the control messages are recorded in the vector clock timestamp $VC^{(k)}$.

We denote the vector clock timestamps of $A^{(k)}\uparrow$ and $A^{(k)}\downarrow$ by $lo^{(k)}$ and $hi^{(k)}$ respectively. The initial timestamp of $A^{(k)}$ is $[null, null]$. In order to possibly achieve early detection, checking messages containing $[lo^{(k)}, null]$ and $[null, hi^{(k)}]$ are considered to be sent right after $A^{(k)}\uparrow$ and $A^{(k)}\downarrow$ respectively.

Observe that $P^{(k)}$ does not need to send checking messages to P_{che} every time $A^{(k)}\uparrow$ or $A^{(k)}\downarrow$ is detected. It only needs to send checking messages once it receives control messages from other non-checker processes. This is mainly because sending control messages will only increase $VC^{(k)}[k]$ for $P^{(k)}$. Redundant increases of $VC^{(k)}[k]$ will not affect the interpretation of the happen-before relation on P_{che}. The $flag\_msg\_act$ (initially $false$) is used to reduce redundant checking messages. If timestamp $[lo^{(k)}, null]$ should not be sent via a checking message, it is stored in the $MsgQue^{(k)}$ of messages to be sent (initially empty) for later use. Pseudo codes of CADA on $P^{(k)}$ are listed in Algorithm 1.

Algorithm 1 CADA on $P^{(k)}$

Upon $A^{(k)}\uparrow$
send_ctl_msg($VC^{(k)}$) to each $P^{(i)}(i \neq k)$;
$lo^{(k)} := VC^{(k)}$;
if $flag\_msg\_act$ = true **then**
  send_chk_msg($[lo^{(k)}, null]$) to P_{che};
else
  store $[lo^{(k)}, null]$ in $MsgQue^{(k)}$;
end if
$++VC^{(k)}[k]$;

Upon $A^{(k)}\downarrow$
$hi := VC^{(k)}$;
if $flag\_msg\_act$ = true **then**
  if $MsgQue^{(k)} \neq empty$ **then**
    restore $[lo^{(k)}, hi^{(k)}]$;
    send_chk_msg($[lo^{(k)}, hi^{(k)}]$) to P_{che};
    clear $MsgQue^{(k)}$;
  else
    send_chk_msg($[null, hi^{(k)}]$) to P_{che};
  end if
  $flag\_msg\_act$:= false;
else
  clear $MsgQue^{(k)}$;
end if

Upon receive_ctl_msg($VC^{(j)}$)
for i = 1 to n **do**
  $VC^{(k)}[i] := max\{VC^{(k)}[i], VC^{(j)}[i]\}$;
end for
$flag\_msg\_act$:= true;

6.2 CADA on the Checker Process Side

P_{che} maintains a queue $Que^{(k)}$ (initially empty) for each $P^{(k)}$. $Que^{(k)}$ stores timestamps from $P^{(k)}$, and P_{che} compares the head elements of the queues to decide whether Formula (2) or Formula (4) is satisfied.

Upon receiving a timestamp via some checking message, P_{che} decides whether comparison of timestamps should be triggered (via subroutine $whether\_trigger\_checking()$). P_{che} first pairs up lo and hi of the same activity. Then it sees whether this timestamp is inserted into some empty queue. If it is, iterative comparison among timestamps (via subroutine $do\_checking()$) is triggered.

In the comparison, timestamps which cannot establish the relations required by Formula (4) or (2) are iteratively deleted. When Formula (2) is satisfied, we first detect whether there exists some process satisfying Formula (4). If there exists such a process, early detection is achieved. Otherwise detection of overlapping among contextual activities is achieved. Pseudo codes of the CADA algorithm on P_{che} are listed in Algorithms 2 and 3.

6.3 Complexity Analysis

Message complexity. As for control messages, the total message cost is $O(n^2p)$, where the number of contextual activities detected by one specific $P^{(k)}$ is $O(p)$, and there are n non-checker processes in total. Please note that the control messages are sent to cope with the asynchrony of pervasive computing environments. Existing context-aware schemes may not need to send control messages. However, they assume the availability of global time, thus not working in asynchronous environments.

Another type of existing work closely related to CADA is the predicate detection algorithm for the debugging of distributed programs [4]. However, predicate detection algorithms passively piggyback over the program being debugged. They do not proactively send control messages among distributed processes. The number of control messages sent are decided by application logic of the program being debugged.

Algorithm 2 CADA on P_{che}

Upon receiving a checking message;
if whether_trigger_checking() = true **then**
  do_checking(); /* Algorithm 3 */
end if

Bool whether_trigger_checking()
Upon receive_chk_msg(Msg) from $P^{(k)}$
$result$:= false;
if $Msg = [lo^{(k)}, hi^{(k)}]$ **then**
  $Que^{(k)}.enqueue([lo^{(k)}, hi^{(k)}])$;
  if head($Que^{(k)}$) = $[lo^{(k)}, hi^{(k)}]$ **then**
    $result$:= true;
  end if
else if $Msg = [lo^{(k)}, null]$ **then**
  $Que^{(k)}.enqueue([lo^{(k)}, null])$;
  if head($Que^{(k)}$) = $[lo^{(k)}, null]$ **then**
    $result$:= true;
  end if
else if $Msg = [null, hi^{(k)}]$ **then**
  combine $[null, hi^{(k)}]$ with $[lo^{(k)}, null]$ at the end of $Que^{(k)}$, and obtain $[lo^{(k)}, hi^{(k)}]$;
  replace $[lo^{(k)}, null]$ with $[lo^{(k)}, hi^{(k)}]$ in $Que^{(k)}$;
  if head($Que^{(k)}$) = $[lo^{(k)}, hi^{(k)}]$ **then**
    $result$:= true;
  end if
end if
Return $result$;

As for checking message, the total message cost is $O(np)$.

Time/space cost. On P_{che}, the number of comparisons is $O(n^2p)$, due to the nested while-for loop. The space cost is also $O(n^2p)$, since there are $O(np)$ queue elements and each element is a vector of n timestamps. On one specific $P^{(k)}$, the time cost is $O(p)$ for the message activities. The space complexity is $O(n)$, for recording the vector clock timestamps.

7 Performance Analysis of CADA in Pervasive Computing Scenarios

To detect the concurrency property in asynchronous environments, CADA mainly depends on the sending/receiving of control messages. However, it is possible that the contextual activities do overlap in a pervasive computing scenario, but the control messages are not received in time due to message delay. Thus, the required happen-before relation is not established, and CADA does not detect the overlapping.

Please note that in this case, CADA is still correct with respect to the semantic interpretation. According to the semantic interpretation, when the happen-before relation required in Formula (2) is not established, CADA should not report detection of the concurrency property, regardless of whether the activities overlap in the physical environment.

Algorithm 3 do_checking()

Case 1: $Msg = [lo^{(k)}, null]$
for i = 1 to n, $i \neq$ k **do**
  $hi^{(i)} := head(Que^{(i)}).hi$;
  while $hi^{(i)} \neq null$ and $\neg(lo^{(k)} \rightarrow hi^{(i)})$ **do**
    delete_head($Que^{(i)}$);
  end while
end for
Case 2: $Msg = [null, hi^{(k)}]$ or $Msg = [lo^{(k)}, hi^{(k)}]$
$changed := \{k\}$;
while $changed \neq \phi$ **do**
  $newchanged := \phi$;
  for all $i \in changed, 1 \leqslant j \leqslant n$ **do**
    if $hi^{(i)} \neq null$ and $\neg(lo^{(j)} \rightarrow hi^{(i)})$ **then**
      $newchanged := newchanged \cup \{i\}$;
    end if
    if $hi^{(j)} \neq null$ and $\neg(lo^{(i)} \rightarrow hi^{(j)})$ **then**
      $newchanged := newchanged \cup \{j\}$;
    end if
  end for
  $changed := newchanged$;
  for all $i \in changed$ **do**
    delete_head($Que^{(i)}$);
  end for
end while

if All activities satisfy Formula (2) **then**
  if Some activity satisfies Formula (4) **then**
    return EARLY-DETECTION;
  end if
  return DETECTION;
end if

On the other hand, however, the case where the overlapping cannot be detected due to the message delay is meaningful to analysis on the performance of CADA in realistic pervasive computing scenarios. In this section, we first define the performance criteria for this case. Then we quantify the expected performance of CADA.

7.1 Performance Criteria

We define $Prob_{det}$ - the *probability for detection of overlapping* - to measure the performance of CADA in pervasive computing scenarios. Specifically, $Prob_{det} = \frac{N_{det}}{N_{phy}}$, where N_{phy} stands for the number of overlapping among contextual activities in the physical environment, and N_{det} denotes the number of overlapping reported by CADA. Similarly, we define the *probability for early detection* $Prob_{early} = \frac{N_{early}}{N_{phy}}$, where N_{early} stands for the number of early detections.

In the ideal case, $Prob_{det}$ should be 100%. However, if we do not have a predetermined bound on the message delay, there is always the probability that the control message cannot build the happen-before relation required by Formula (2). We can see this from

the following Fig. 1. Without the bound on message delay, $msg^{(2)}$ always has the probability to arrive at $P^{(1)}$ after $A^{(1)}\downarrow$. In this case, P_{che} should not report the satisfaction of $Def(\phi)$ since Formula (2) is not satisfied. However, the activities actually overlap in the physical environment.

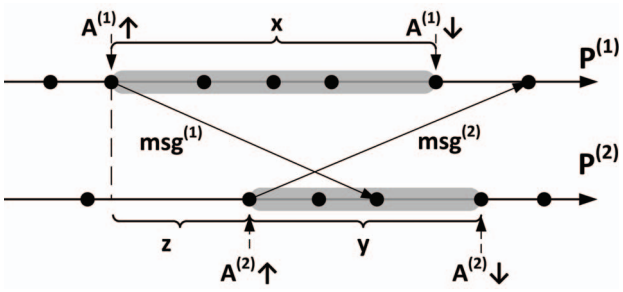


Fig. 1. Failure of overlapping detection.

7.2 Impact of Message Delay on Performance of CADA

In this section, we quantify the relation between message delay and the performance of CADA. In our analysis, we first focus on the case of two non-checker processes. Then we discuss the case of more than 2 processes.

7.2.1 Impact of Message Delay on $Prob_{det}$

To quantify the impact, we first analyze when and how the required happen-before relation is not established due to message delay. Observe that when CADA detects overlapping among contextual activities, the activities do overlap in the physical environment. This is mainly because when the activities do not overlap, the happen-before relation required in Formula (2) cannot be established, now matter how quickly the control messages arrive.

However, when the activities do overlap in the physical environment, CADA may not detect this overlapping if the required happen-before relation is not established due to the delay of control messages, as shown in Fig. 1. The happen-before relation required in Formula (2) is established if and only if $msg^{(1)}$ and $msg^{(2)}$ are received in time. Formally,

$$Prob_{det} = P(x,y,z) = Pr\{msg^{(1)}.delay \leqslant y+z\} \quad (5)$$
$$Pr\{msg^{(2)}.delay \leqslant x-z\}$$

Here we define random variables X, Y and Z, whose values are x, y and z respectively, as shown in Fig. 1. The expected probability for detection of activity overlapping is:

$$E[Prob_{det}] = E[P(x,y,z)] = \int_0^\infty \{\int_0^\infty [\int_0^x P(x,y,z) dF_3(z)] dF_2(y)\} dF_1(x)$$

Here, the probability functions $F_1(x) = Pr\{X \leqslant x\}$, $F_2(y) = \{Y \leqslant y\}$, $F_3(z) = Pr\{Z \leqslant z\}$.

To further investigate $E[Prob_{det}]$, we model the duration of $A^{(k)}$ based on the queuing theory [5], [6]. Specifically, a queue of intervals with Poisson arrival rate $\lambda^{(k)}$ is adopted. The duration of intervals follows the exponential distribution of rate $\mu^{(k)}(\frac{\lambda^{(k)}}{\mu^{(k)}} \leqslant 1)$. We also need to model the message delay. 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. Although 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 [7]. In our analysis, we adopt the exponential distribution to model message delay. Note that our analysis is also applicable when message delay follows other types of distributions. With the models we adopt, we have that:

$$Prob_{det} = P(x,y,z) = (1-e^{-\lambda(y+z)})(1-e^{-\lambda(x-z)})$$

According to [6], the probability function of the activity duration is:

$$\begin{aligned} F_1(x) &= Pr\{X \leqslant x\} \\ &= \int_0^x \sum_{k=1}^{\infty} \frac{\mu^{(1)}(\lambda^{(1)}\mu^{(1)}x^2)^{k-1}}{k!(k-1)!} e^{-(\lambda^{(1)}+\mu^{(1)})x} dx \end{aligned}$$

$$\begin{aligned} F_2(y) &= Pr\{Y \leqslant y\} \\ &= \int_0^y \sum_{k=1}^{\infty} \frac{\mu^{(2)}(\lambda^{(2)}\mu^{(2)}y^2)^{k-1}}{k!(k-1)!} e^{-(\lambda^{(2)}+\mu^{(2)})y} dy \end{aligned}$$

Since the arrival of $A^{(2)}$ follows the Poisson process, we have that, for $z > 0$:

$$F_3(z) = Pr\{Z \leqslant z\} = 1 - e^{-\lambda^{(2)}z}$$

To illustrate the performance of CADA in terms of $Prob_{det}$, we change the average message delay and obtain the numerical results, as shown in Fig. 2. We choose to vary the average message delay because it has the most significant impact on $E[Prob_{det}]$. According to the numerical results, we find that $E[Prob_{det}]$ remains near 100% when the message delay is not quite long. As the average message delay significantly increases, $E[Prob_{det}]$ decreases quite slowly (note that the x-axis is the logarithm of message delay). This ensures that CADA achieves satisfying performance in terms of $Prob_{det}$ in various pervasive computing environments with different message delay.

7.2.2 Impact of Message Delay on $Prob_{early}$

In context-aware applications, it is quite desirable, if not a must, to detect specified concurrency property as early as possible. However, this issue was not sufficiently addressed in the existing work, e.g., in debugging of distributed programs[8], [9], [4]. In existing schemes, P_{che} only receives timestamps of a complete

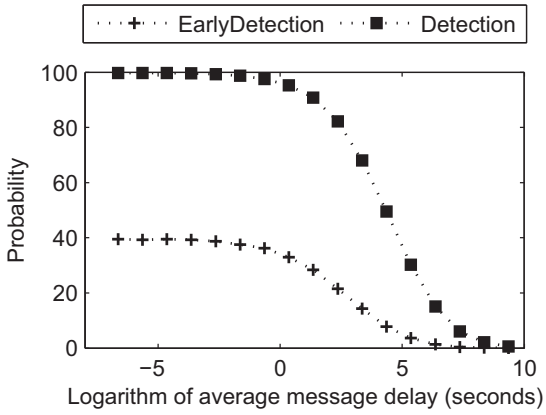


Fig. 2. Probabilities $Prob_{det}$ and $Prob_{early}$.

activity (with both lo and hi). Thus P_{che} detects the overlapping no earlier than the latest hi among all activities [4]. CADA detects the overlapping in a finer granularity. The timestamp of lo or hi is sent right after the $\uparrow$ or $\downarrow$ of $A^{(k)}$. Thus, CADA has chances to detect the overlapping earlier.

We first discuss when and how CADA achieves early detection. As shown in Fig. 3, if the overlapping is checked only when both lo and hi of the activities are collected, we detect concurrent activities after $A^{(2)}\downarrow$. However, we can still detect that $A^{(1)}$ and $A^{(2)}$ overlap without the timestamp of $A^{(2)}\downarrow$. Observe that if we guarantee:

$$(A^{(1)}.lo \rightarrow A^{(2)}.lo) \wedge (A^{(2)}.lo \rightarrow A^{(1)}.hi) \wedge (y > x - z)$$

we guarantee that $A^{(1)}$ and $A^{(2)}$ overlap. In this case, CADA detects concurrent activities after $A^{(1)}.hi$ is collected, not having to wait for $A^{(2)}.hi$.

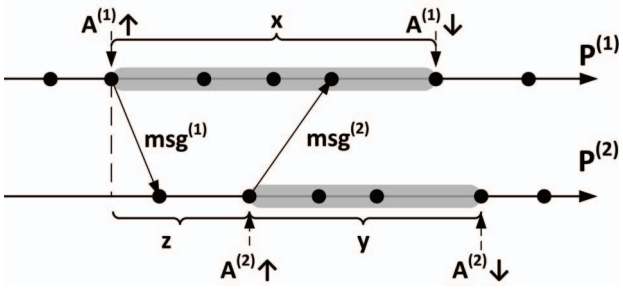


Fig. 3. CADA achieving early detection.

Based on the analysis above, we have that:

$$\begin{aligned} Prob_{early} &= P'(x,y,z) \\ &= Pr\{msg^{(1)}.delay < z\} \\ &\quad Pr\{msg^{(2)}.delay < x - z\} Pr\{y > x - z\} \\ &= (1 - e^{-\lambda z})(1 - e^{-\lambda(x-z)})(1 - F_2(x - z)) \end{aligned} \quad (6)$$

The expected probability of early detection is:

$$E[Prob_{early}] = \int_0^\infty \{\int_0^\infty [\int_0^x P'(x,y,z) \, dF_3(z)] dF_2(y)\} dF_1(x)$$

We also illustrate performance of CADA in terms of $E[Prob_{early}]$ by numerical results. From Fig. 2, we find that $E[Prob_{early}]$ is around half of $E[Prob_{det}]$ when the message delay is not quite long and both probabilities remain stable. Both probabilities decrease in a similar way as the average message delay significantly increases, i.e., $E[Prob_{early}]$ can also tolerate significant increase in the average message delay. Thus CADA achieves satisfying performance in terms of both probabilities in environments with different message delay.

7.3 Discussions

In the analysis above, we mainly focus on the case of 2 non-checker processes. Analytical results for the case of n ($\geqslant 3$) processes can also be derived based on the results obtained above. Specifically, we first define some notations. Let duration of the contextual activity on the k^{th} non-checker process be $l^{(k)}$, starting from $start^{(k)}$ and ending at $end^{(k)}$. For the ease of interpretation, we assume that the contextual activities are sorted based on $start^{(k)}$, i.e., $start^{(1)} \leqslant start^{(2)} \leqslant \cdots \leqslant start^{(n)}$.

We obtain $Prob_{det}$ by calculating the probability that each pair of contextual activities overlap. Given the analytical result for one pair of processes in Formula (5), we have that:

$$\begin{aligned} &Prob_{det}(1, \cdots, n) \\ &= \prod_{1 \leqslant i < j \leqslant n} P(l_i, l_j, start^{(j)} - start^{(i)}) \end{aligned} \quad (7)$$

Similar to the analysis in Section 7.2.1, we can further compute the expected probability $E[Prob_{det}(1, \cdots, n)]$ by integration over variables $l_k (1 \leqslant k \leqslant n)$ and $start^{(j)} - start^{(i)} (1 \leqslant i < j \leqslant n)$. Here, l_k varies in $(0, \infty)$, and $start^{(j)} - start^{(i)}$ varies in $(0, l_i)$.

As for $Prob_{early}(1, \cdots, n)$, the case of early detection is that all the contextual activities on the first $n-1$ processes $P^{(1)}, P^{(2)}, \cdots P^{(n-1)}$ overlap, while the contextual activity on $P^{(n)}$ satisfies Formula (4) with every other contextual activity. Specifically, based on the analytical results in Formula (6) and (7), we have that:

$$\begin{aligned} &Prob_{early}(1, \cdots, n) \\ &= Prob_{det}(1, \cdots, n-1) \\ &\quad \prod_{1 \leqslant i \leqslant n-1} P'(l_i, l_n, start^{(n)} - start^{(i)}) \end{aligned}$$

The expected probability $E[Prob_{early}(1, \cdots, n)]$ can also be computed by integration over variables $l_k (1 \leqslant k \leqslant n)$ and $start^{(n)} - start^{(i)} (1 \leqslant i \leqslant n-1)$. Here, l_k varies in $(0, \infty)$, and $start^{(n)} - start^{(i)}$ varies in $(0, l_i)$.

8 Performance Measurements

8.1 Implementation

The detection of contextual properties assumes the availability of an underlying context-aware middleware. We have implemented the middleware based on one of our research projects - *Middleware Infrastructure for Predicate detection in Asynchronous environments* (MIPA) [10], [11]. The system architecture of MIPA is shown in Fig. 4.

From MIPA's point of view, the application achieves context-awareness by specifying contextual properties of its interest to MIPA. The checker process is implemented as a third-party service, plugged into MIPA. Non-checker processes are deployed to manipulate context collecting devices, monitoring different parts of the environment.

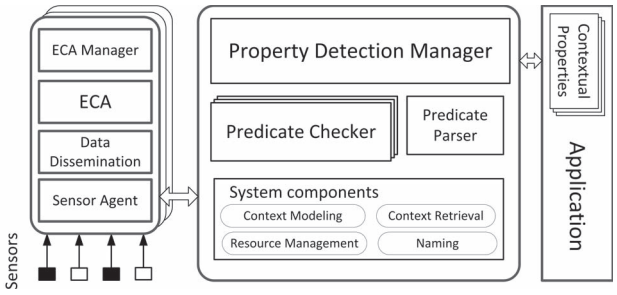


Fig. 4. System architecture of MIPA.

8.2 Experiment Configurations

In the performance measurements, user's stay inside/outside the meeting room and the working periods of the projector are generated following the queues discussed in our analysis of CADA in pervasive computing scenarios (Section 7). We model the message delay by exponential distribution, and set different intervals between context updates on the mobile phone.

In the experiments, we study $Prob_{det}$ and $Prob_{early}$ (defined in Section 7). We first vary the asynchrony (message delay and interval between context updates) of the environment. We also vary the duration of contextual activities. Finally, we tune the number of non-checker processes, which complements our analysis in Section 7.

The lifetime of experiment is set to 500 hours, and on each non-checker process, around 2,000 contextual activities are generated. We average the evaluation results over 300,000 runs to make the results statistically stable. Detailed experiment configurations are listed in Table 2.

TABLE 2
Experiment configurations

Parameter	Value
Lifetime of experiment	500 hours
Interval between local predicate update	1min
Average duration of/between contextual activities	10min/5min
Average message delay	0s to 10s
Average interval between context updates	1min to 100min
Range for tuning duration of contextual activities	[5min, 50min]
Number of non-checker processes	2 to 8

8.3 Evaluation Results

8.3.1 Effects of Tuning the Message Delay

In this experiment, we study how the message delay affects the performance of CADA. We fix the interval between context updates to 1min, and let the non-checker processes update local predicates every 1min. We find that when encountered with reasonably long message delay (less than 10s), $Prob_{det}$ is quite high (a little less than 100%) and $Prob_{early}$ is nearly half of $Prob_{det}$, as shown in Fig. 5. The evaluation results are in accordance with our performance analysis (Section 7). The message delay results in monotonic decrease of $Prob_{det}$ and $Prob_{early}$, mainly due to the increase in the asynchrony of the environment.

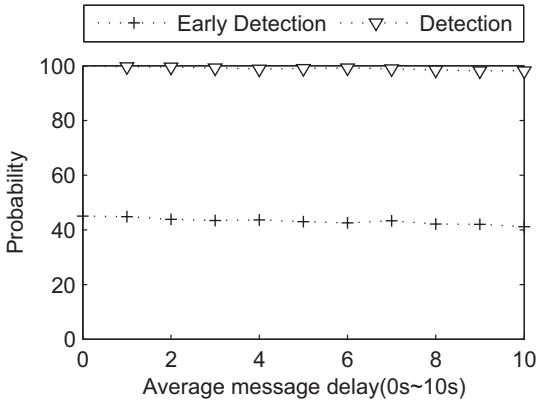


Fig. 5. Effects of Tuning the Message Delay.

8.3.2 Effects of Tuning the Interval between Context Updates

In this experiment, we study how the interval between context updates affects the performance of CADA. To focus on the impact of the interval, the message delay is set to zero in this experiment. We first study the impact when the interval is no more than 10min (which is the average duration of contextual activities). We find that $Prob_{early}$ is around half of $Prob_{det}$. The increase in the interval results in monotonic decrease in $Prob_{det}$ and $Prob_{early}$ by 16.6% and 13.3% respectively, as shown in Fig. 6.

Then we greatly increase the interval to 100min and see how $Prob_{det}$ and $Prob_{early}$ decrease. We find that both probabilities decrease quickly as the interval greatly increases. When the interval increases to 60min, the decreases in both probabilities s-

low down. When we increase the update interval to around 100min, $Prob_{det}$ decreases to around 10% and $Prob_{early}$ decreases to around zero, as shown in Fig. 7.

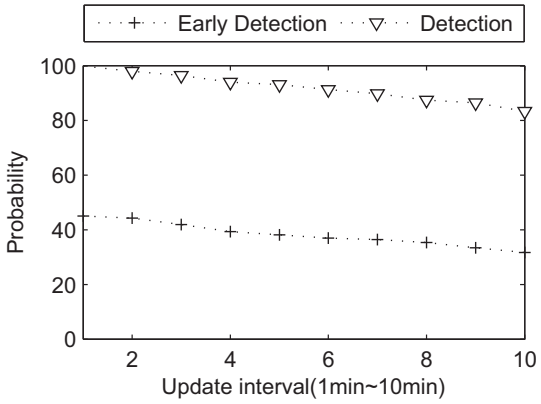


Fig. 6. Effects of Tuning the Update Interval.

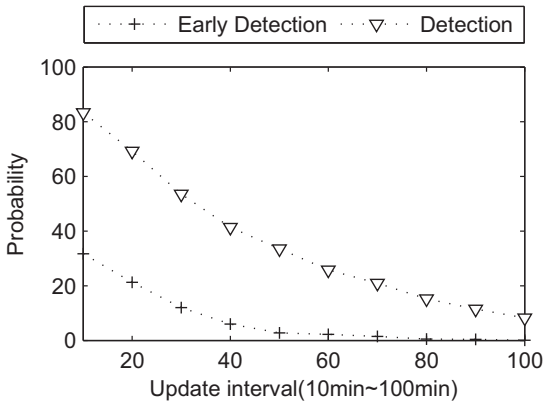


Fig. 7. Effects of Tuning the Update Interval.

8.3.3 Effects of Tuning the Activity Duration

In this experiment, we study the impact of the average duration of contextual activities. The average duration is tuned from 5min to 50min. We fix the message delay to 0.5s and the interval between context updates to 3min. We find that $Prob_{det}$ slightly increases (by around 4.4%), while $Prob_{early}$ decreases (by around 31.6%) when the duration increases, as shown in Fig. 8. $Prob_{det}$ increases mainly because with longer activities, the control messages have more chances to arrive in time. As for $Prob_{early}$, duration of the activities do not impact whether $msg^{(1)}$ arrive in time, as shown in Fig. 3. Moreover, long activities ($x > z + y$) could reduce the chances of early detection. Thus $Prob_{early}$ decreases.

This is in accordance with our analysis in Section 7. As shown in the derivation of $E[Prob_{det}]$, when the duration of activities increases, both x and y increase, resulting in increase in $Prob_{det}$. The decrease in $Prob_{early}$ mainly results from the increase in $F_2(x - z)$, as shown in the derivation of $E[Prob_{early}]$. Since $F_2(x - z)$ is a probability function and is thus monotonically non-decreasing, increase in x results in the increase in $F_2(x - z)$.

Overall, impact of the average duration of contextual activities on $Prob_{det}$ and $Prob_{early}$ is not significant, which makes CADA widely applicable to detect concurrent activities of different duration.

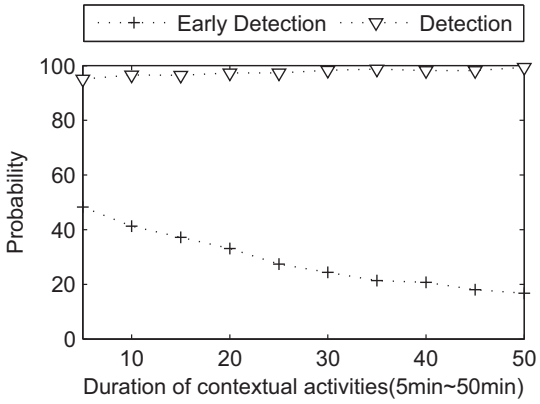


Fig. 8. Effects of Tuning the Duration of Contextual Activities

8.3.4 Effects of Tuning the Number of Non-checker Processes

We also study the impact of the number of non-checker processes, which complements our analysis in Section 7. We fix the interval between context updates and the message delay to 3min and 0.5s respectively. We set average duration of contextual activities to 10min.

As the number of non-checker processes increases, $Prob_{det}$ linearly decreases and $Prob_{early}$ decreases a little faster, as shown in Fig. 9. This is mainly because, for multiple activities, any activity not following Formula (2) will falsify $Def(\phi)$, i.e., the asynchrony of coordination among non-checker processes accumulates as the number of non-checker processes increases.

9 Additional Discussions on the Related Work

Our work can be positioned against two areas of existing work: context-aware computing and detection of global predicates in distributed computations.

As for context-aware computing, in [12], properties were modeled by tuples, and property detection was based on comparison among elements in the tuples. In [13], contextual properties were expressed in first-order-logic, and an incremental property detection

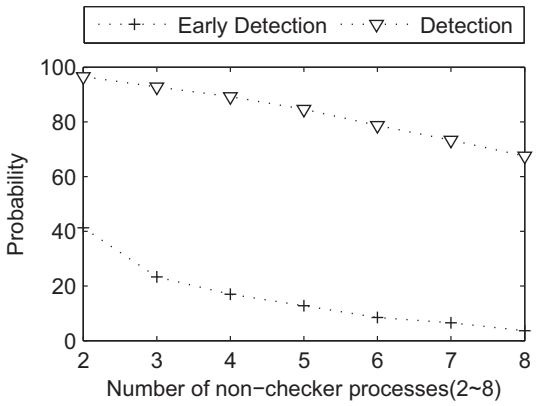


Fig. 9. Effects of Tuning the Number of Non-checker Processes

algorithm was proposed. In the existing work, temporal relations among the collected contexts are not sufficiently considered.

In asynchronous computations, the concept of time must be carefully reexamined [1], [14], and logical clocks are devised to cope with the asynchrony [2]. In [15], Chandy et al. studied how to obtain a snapshot, i.e., a meaningful observation, of an asynchronous computation, and then detect stable predicates based on the snapshots. In [8], Cooper et al. investigated the detection of unstable predicates, which brought combinatorial explosion of the state space. Conjunctive predicates play a key role in detection of unstable predicates and conjunctive predicates under different modalities were studied in [9], [4]. In [14], [16], Kshemkalyani et al. gave a refinement of the traditional coarse-grained modality classification and corresponding predicate detection algorithms are proposed.

A constructive proof is presented in [4], which also builds the equivalence between $Def(\phi)$ and the overlapping among contextual activities. Comparatively, in our work, the semantic interpretation of $Def(\phi)$ is different. It utilizes the lattice structure among all CGSs of the pervasive computing environment, and this lattice structure is used in our proof of the equivalence between $Def(\phi)$ and the overlapping among contextual activities. We argue that the semantic interpretation based on the lattice of CGSs is more general and can facilitate further investigation of detecting global predicates in asynchronous pervasive computing environments (e.g., in [17]). Moreover, it is not studied in [4] whether the predicate detection reflects the actual status of contextual activities in the physical environment and whether timely detection is achieved. However, these issues are important for pervasive computing scenarios, and we conduct both theoretical analysis and experimental evaluation to address these issues for CADA.

REFERENCES

[1] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[2] F. Mattern, "Virtual time and global states of distributed systems," in *Proc. International Workshop on Parallel and Distributed Algorithms*, Holland, 1989, pp. 215–226.

[3] R. Schwarz and F. Mattern, "Detecting causal relationships in distributed computations: in search of the holy grail," *Distrib. Comput.*, vol. 7, no. 3, pp. 149–174, 1994.

[4] V. K. Garg and B. Waldecker, "Detection of strong unstable predicates in distributed programs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, pp. 1323–1333, Dec. 1996.

[5] B. Bunday, *An Introduction to Queueing Theory*. London: A Hodder Arnold Publication, 1996.

[6] Y. Tang and X. Tang, *Queuing Theory: Fundamentals and Analysis Techniques (in Chinese)*. Beijing, China: Science Press, 2006.

[7] N. Duffield and F. Lo Presti, "Multicast inference of packet delay variance at interior network links," in *Proc. IEEE Conference on Computer Communications (INFOCOM'00)*, Tel Aviv, Israel, Mar 2000, pp. 1351–1360 vol.3.

[8] R. Cooper and K. Marzullo, "Consistent detection of global predicates," in *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, New York, NY, USA, 1991, pp. 167–174.

[9] V. Garg and B. Waldecker, "Detection of weak unstable predicates in distributed programs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, pp. 299–307, Mar. 1994.

[10] "MIPA - Middleware Infrastructure for Predicate detection in Asynchronous environments." [Online]. Available: http://mipa.googlecode.com

[11] J. Yu, Y. Huang, J. Cao, and X. Tao, "Middleware support for context-awareness in asynchronous pervasive computing environments," in *Proc. International Conference on Embedded and Ubiquitous Computing (EUC'10)*, Hong Kong, China, Dec. 2010.

[12] C. Xu and S. C. Cheung, "Inconsistency detection and resolution for context-aware middleware support," in *Proc. ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'05)*, Lisbon, Portugal, Sep. 2005, pp. 336–345.

[13] C. Xu, S. C. Cheung, and W. K. Chan, "Incremental consistency checking for pervasive context," in *Proc. International Conference on Software Engineering (ICSE'06)*, Shanghai, China, May 2006, pp. 292–301.

[14] A. D. Kshemkalyani, "A fine-grained modality classification for global predicates," *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, no. 8, pp. 807–816, 2003.

[15] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, pp. 63–75, February 1985.

[16] P. Chandra and A. D. Kshemkalyani, "Causality-based predicate detection across space and time," *IEEE TRANSACTIONS ON COMPUTERS*, vol. 54, no. 11, pp. 1438–1453, 2005.

[17] T. Hua, Y. Huang, J. Cao, and X. Tao, "A lattice-theoretic approach to runtime property detection for pervasive context," in *Proc. International Conference on Ubiquitous Intelligence and Computing (UIC'10)*, Xian, China, Oct. 2010, pp. 307–321.