Formal Specification and Runtime Detection of Dynamic Properties in Asynchronous Pervasive Computing Environments

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

Abstract—Formal specification and runtime detection of contextual properties is one of the primary approaches to enabling context awareness in pervasive computing environments. Due to the intrinsic dynamism of the pervasive computing environment, dynamic properties, which delineate concerns of context-aware applications on the temporal evolution of the environment state, are of great importance. However, detection of dynamic properties is challenging, mainly due to the intrinsic asynchrony among computing entities in the pervasive computing environment. Moreover, the detection must be conducted at runtime in pervasive computing scenarios, which makes existing schemes do not work. To address these challenges, we propose the property detection for asynchronous context (PDAC) framework, which consists of three essential parts: 1) Logical time is employed to model the temporal evolution of environment state as a lattice. The *active surface* of the lattice is introduced as the key notion to model the runtime evolution of the environment state; 2) Specification of dynamic properties is viewed as a formal language defined over the trace of environment state evolution; and 3) The *SurfMaint* algorithm is proposed to achieve runtime maintenance of the active surface of the lattice, which further enables runtime detection of dynamic properties. A case study is conducted to demonstrate how the PDAC framework enables context awareness in asynchronous pervasive computing scenarios. The SurfMaint algorithm is implemented and evaluated over MIPA—the open-source context-aware middleware we developed. Performance measurements show the accuracy and cost-effectiveness of SurfMaint, even when faced with dynamic changes in the asynchronous pervasive computing environment.

Index Terms—Dynamic property, context awareness, asynchrony, predicate detection, lattice

1 Introduction

Pervasive computing demands that applications are capable of operation in highly dynamic environments without explicit user intervention [14], [31]. Thus, pervasive applications are typically context aware, to minimize user intervention [8], [15].

To achieve context awareness, pervasive applications need to be aware of whether contexts bear specified properties, thus being able to adapt their behavior accordingly [30], [37], [38], [26], [39]. For example in an elderly care scenario, the application may specify a property C_1: *the elder has been having meals, and then he tries to take medicine*, and send the elder an alarm that the medicine should be taken before meals, when the property C_1 holds.

Among various contextual properties the application may specify, *dynamic properties*, which delineate the temporal evolution of the environment state, are of great importance. This is mainly due to the intrinsic dynamism of the pervasive computing environment. To cope with this dynamism, the context-aware application needs to keep a stretch of environment state evolution in sight, and use dynamic properties to delineate its concerns about the evolution. In the example above, the application is not concerned about specific status of the elder (e.g., "have meal" or "take medicine"). Rather, it is concerned with the temporal evolution of the elder's status that he was first having meals, and tries to take medicine right after the meal.

However, the specification and detection of dynamic properties is challenging, mainly due to that the computing entities in the pervasive computing environment coordinate in an asynchronous way [8], [17], [18], [20], [21], [23], [32], [40], and that the detection of contextual properties should be effectively conducted at runtime [18], [20]. Specifically, context collecting devices may not have global clocks and may run at different speeds. They heavily rely on wireless communications, which suffer from finite but arbitrary delay. Moreover, context collecting devices (usually battery-powered sensors) may postpone the dissemination of context data due to resource constraints, which also results in asynchrony.

Most existing schemes for detection of contextual properties implicitly assume that context collecting devices share the same notion of time, thus dynamic properties can be detected easily [19], [38]. But this assumption does not necessarily hold in asynchronous pervasive environments. Though a few schemes have been proposed to cope with the

- *Y. Yang, Y. Huang, X. Ma, and J. Lu are with the State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, 163# Xianlin Street, Qixia District, Nanjing, Jiangsu Province, China 210046. E-mail: {csylyang, csyuhuang}@gmail.com, {xxm, 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 7 Mar. 2012; revised 26 July 2012; accepted 31 July 2012; published online 4 Sept. 2012.
Recommended for acceptance by R. Baldoni.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2012-03-0259.
Digital Object Identifier no. 10.1109/TPDS.2012.259.

1045-9219/13/$31.00 © 2013 IEEE Published by the IEEE Computer Society

asynchrony in detection of contextual properties [17], [18], [20], they mainly discuss detection of static properties, i.e., properties concerning specific environment state. Though static properties can capture interesting aspect of the environment, they inherently lack the notion of relative temporal order [4], [23]. Such properties cannot delineate temporal evolution of the environment state.

Moreover, the detection of contextual properties should be conducted at runtime in pervasive computing scenarios. Existing schemes are mainly designed for offline checking (e.g., in debugging of distributed programs [9], [34]) where all the traces of system evolution can be obtained in advance. This assumption does not hold in pervasive computing scenarios, where the observation on environment state evolution is continuous. This imposes challenges on both specification and detection of contextual properties.

Discussions above necessitate a systematic scheme for formal specification and runtime detection of dynamic properties in asynchronous pervasive environments. Toward this objective, we propose the property detection for asynchronous context (PDAC) framework, which consists of three essential parts:

1. *Modeling of the temporal evolution of environment state.* Logical time [10], [24], [28] is employed to capture the temporal evolution of environment state despite of asynchrony. Under logical time, all meaningful environment states compose a lattice [3], [33]. To support runtime observation of the environment, one key notion is the *active surface* of the currently observed lattice. Environment state evolution is modeled as sequences of environment states ending at the active surface.
2. *Specification of dynamic properties.* Boolean predicates specified over specific environment states are regarded as an alphabet. Dynamic properties are viewed as words following the regular grammar over the alphabet. The satisfaction of dynamic properties is interpreted at runtime over the trace of environment state evolution induced by the current active surface.
3. *Detection of the specified dynamic property.* We propose the *SurfMaint* algorithm, which *maint*ains the active *surf*ace at runtime and further enables runtime detection of dynamic properties without maintaining the whole lattice.

A case study of an elderly care scenario is conducted to demonstrate how the PDAC framework supports context-awareness in pervasive computing scenarios. The SurfMaint algorithm is implemented and evaluated over MIPA—the open-source context-aware middleware we developed [1], [20], [40]. Performance measurements show that SurfMaint detects dynamic properties accurately and cost-effectively, even when faced with dynamic changes in the asynchronous pervasive computing environment.

The remainder of this paper is organized as follows: In Section 2, we briefly outline the PDAC framework. In Sections 3, 4, and 5, we discuss the three essential parts of PDAC. In Sections 6 and 7, we present the case study and performance measurements. In Section 8, we review the existing work. Section 9 concludes the paper with a brief summary and the future work.

TABLE 1
Notations in the PDAC Framework

Notation	Explanation
n	number of non-checker processes
$P^{(k)}, P_{che}$	non-checker / checker process ($1 \leqslant k \leqslant n$)
$e_i^{(k)}, s_i^{(k)}$	contextual event / local state on $P^{(k)}$
$\mathcal{G}$	global state
$\mathcal{C}, \mathcal{C}[k]$	CGS, k^{th} constituent local state
$\mathcal{S}(\mathcal{C}_i, \mathcal{C}_j)$	a CGS sequence from $\mathcal{C}_i$ to $\mathcal{C}_j$
$Que^{(k)}$	queue of local states from each $P^{(k)}$ on P_{che}
LAT	lattice of CGSs
$Surf(LAT)$	surface of lattice LAT
$Act(LAT)$	active surface CGSs of lattice LAT
$Evol(LAT)$	all possible CGS sequences of environment state evolution
Σ	alphabet used to label CGSs
$\lambda(\mathcal{C})$	labeling of CGS $\mathcal{C}$
$\bar{\lambda}(\mathcal{S}(\mathcal{C}_i, \mathcal{C}_j))$	labeling of CGS sequence $\mathcal{S}(\mathcal{C}_i, \mathcal{C}_j)$
$\mathcal{L}(LAT)$	language associated with LAT
$Pos(\cdot)/Def(\cdot)$	modal operators
Φ	specified regular expression
Φ'	extension of Φ, used for property detection
$\mathcal{L}(\Phi)$	languages defined by Φ
$\mathcal{A}(\Phi)$	automaton which accepts $\mathcal{L}(\Phi)$
$\mathcal{R}^{\Phi}(\mathcal{C})$	reachable states of CGS $\mathcal{C}$ on $\mathcal{A}(\Phi)$
$\mathcal{R}^{\Phi}(Act(LAT))$	reachable states of $Act(LAT)$ on $\mathcal{A}(\Phi)$

2 Property Detection for Asynchronous Context

In this section, we first describe the system model. Then we outline the PDAC framework. Some elements of the framework have appeared in our previous work [20]. However, previous work only supports the detection of concurrency properties. In this paper, we complete the framework by supporting the detection of dynamic properties. Notations used in the framework are listed in Table 1.

2.1 System Model

The detection of contextual properties assumes the availability of an underlying context-aware middleware [25], [38], [40]. The middleware receives contextual properties from the applications, persistently collects context data from multiple sources that provide the context required by the properties, and detects at runtime whether the properties hold. Specifically, a collection of *nonchecker processes* $P^{(1)}, P^{(2)}, \ldots, P^{(n)}$ are deployed to monitor specific regions or aspects of the environment. Examples of nonchecker processes are software processes manipulating networked sensors. One *checker process* P_{che} is in charge of the detection of contextual properties. P_{che} is usually a third-party service deployed on the middleware.

2.1.1 Asynchronous Message Passing and Logical Time

We model the 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 the FIFO manner [12], [13], [18], [20].

While monitoring specific aspects of the environment, each $P^{(k)}$ generates its (potentially infinite) trace of *local states*

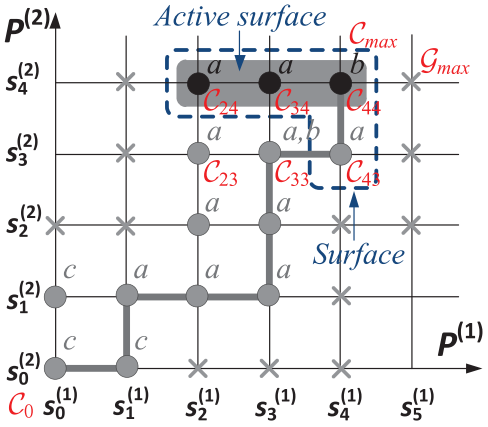

Fig. 1. Lattice of CGSs.

connected by *contextual events*: "$e_0^{(k)}, s_0^{(k)}, e_1^{(k)}, s_1^{(k)}, \ldots$." Contextual events may be local, for example, update of context data, or global, for example, sending/receiving messages. We re-interpret the notion of time based on Lamport's definition of the *happen-before* relation (denoted by "$\rightarrow$") resulting from message causality [24], as well as the coding of this happen-before relation by the logical vector clock [28]. Detailed definitions 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.2012.259.

2.1.2 Lattice of Consistent Global States (CGSs)

The notion of CGS is essential in our system model. Specifically, a *global state* $\mathcal{G} = (s^{(1)}, s^{(2)}, \ldots, s^{(n)})$ is defined as a vector of local states from each $P^{(k)}$. If the constituent states of a global state $\mathcal{C}$ are pairwise concurrent, $\mathcal{C}$ is a CGS, i.e.,

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

A CGS denotes a snapshot or meaningful observation of the asynchronous environment state [3], [33].

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

$$\mathcal{C} \prec \mathcal{C}' \stackrel{def}{=\!=} \exists\, k, \text{ such that } \mathcal{C}'[k] \text{ is the first local state after } \mathcal{C}[k] \text{ on } P^{(k)}, \forall\, l : l \neq k :: \mathcal{C}[l] = \mathcal{C}'[l].$$

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

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

One key notion is that the set of observed CGSs with the "$\rightsquigarrow$" relation has the lattice structure (denoted by LAT) [3], [33]. Fig. 1 shows a currently observed lattice. The dots denote CGSs and edges depict the "$\prec$" relation. Crosses "$\times$" denote inconsistent global states.

To model the temporal evolution of the environment state, we need to keep a stretch of meaningful observations (i.e., CGSs) in sight. Therefore, we define *CGS sequences* in the lattice. A CGS sequence $\mathcal{S}(\mathcal{C}_i, \mathcal{C}_j)$ is a sequence of CGSs connected by "$\prec$," i.e.,

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

For example, in Fig. 1, the bold broken line indicates a CGS sequence starting from $\mathcal{C}_0$ and ending at $\mathcal{C}_{44}$.

2.2 Elements of PDAC

Here, we outline the three essential parts of the PDAC framework. They are discussed in detail in the following Sections 3, 4, and 5.

2.2.1 Modeling of the Temporal Evolution of Environment State

While persistently monitoring the environment, the checker process observes temporal evolution of the environment at runtime. As discussed in Section 2.1.2, the meaningful observations (CGSs) of environment state form a lattice. More importantly, as the environment is continuously changing, the observed lattice "grows" at runtime, to a potentially infinite size.

In this continuously growing lattice, we are most concerned with *active CGSs* (e.g., black CGSs in the gray rectangle in Fig. 1) which could have new immediate successor CGSs in the future. In a geometrical view, active CGSs are in the *surface* (e.g., the dotted-line L-shaped polygon in Fig. 1) of the lattice. We are concerned with *active surface*, because all further growth of CGSs stems from it. Consequently, the continuous changing of the environment, i.e., the continuous growing of the lattice, can be characterized by the updating of the active surface at runtime.

Active-surface-induced CGS sequences, i.e., CGS sequences that originate from the initial CGS and span to active surface CGSs, model the ongoing environment state evolution. They are observed finite prefixes of potentially infinite environment state evolutions. The bold broken line in Fig. 1 is one of the active-surface-induced CGS sequences. Furthermore, the labeling of active-surface-induced CGS sequences forms the *trace of environment state evolution*.

2.2.2 Specification of Dynamic Properties

Specification of dynamic properties is defined over the active-surface-induced CGS sequences defined above. Specifically, Boolean predicates specified over specific CGSs are regarded as an alphabet. Dynamic properties are viewed as words following the regular grammar over the alphabet.

We use regular expressions because they have rich semantics for context-aware applications to describe their concerns on the temporal evolution of the environment state. Applications can specify regular patterns related to counting that cannot be expressed using temporal logics, for example, $(0(0+1))^*$ saying that every other letter is 0 [35]. Meanwhile, manipulation of regular expressions is relatively more tractable.

To persistently monitor the environment at runtime is to check whether the currently observed finite prefixes of the

(potentially infinite) environment state evolution satisfy the specified dynamic property. Thus, we interpret the semantics over the observed (finite) active-surface-induced CGS sequences.

2.2.3 Detection of the Specified Dynamic Property

Based on the modeling and specification above, detection of dynamic properties is reduced to the online recognition of regular expressions over the trace of environment state evolution.

Specifically, the detection is conducted in three steps: 1) the specified regular expression is transformed to the equivalent deterministic finite automaton (DFA); 2) the active surface is maintained at runtime; and 3) the trace of environment state evolution is injected into the DFA at runtime. When reaching an accepting state in the DFA, the property is detected.

3 Modeling of the Temporal Evolution of Environment State

In this section, we first define the active surface as well as the active-surface-induced CGS sequences. Then, we discuss the trace of environment state evolution.

3.1 Active Surface and Active-Surface-Induced CGS Sequences

As discussed in Section 2.2.1, to capture runtime evolution of the environment, we are concerned with the surface CGSs in the lattice. Informally, surface CGSs are on the boundary of the currently observed lattice. To formally define the surface, first note that P_{che} uses a queue $Que^{(k)}(1 \leq k \leq n)$ to store the local states (in FIFO manner) sent from each $P^{(k)}$ [13], [17], [20]. Let LAT be the currently observed lattice, $\mathcal{C}_{max}$ be the maximal CGS, and $\mathcal{G}_{max}$ be the maximal global state (not necessarily consistent). For example, in Fig. 1, the current $Que^{(1)} = \{s_0^{(1)}, s_1^{(1)}, s_2^{(1)}, s_3^{(1)}, s_4^{(1)}, s_5^{(1)}\}$, $Que^{(2)} = \{s_0^{(2)}, s_1^{(2)}, s_2^{(2)}, s_3^{(2)}, s_4^{(2)}\}$, $\mathcal{C}_{max} = (s_4^{(1)}, s_4^{(2)})$, and $\mathcal{G}_{max} = (s_5^{(1)}, s_4^{(2)})$.

Definition 1 (Surface). *Given the observed lattice LAT and the maximal CGS $\mathcal{C}_{max}$, the surface of LAT is defined as follows:*

$$Surf(LAT) = \{\mathcal{C} | \mathcal{C} \in LAT, \exists k, \mathcal{C}[k] = \mathcal{C}_{max}[k]\}.$$

Refer to the geometric illustration in Fig. 1, $Surf(LAT) = \{\mathcal{C}_{24}, \mathcal{C}_{34}, \mathcal{C}_{43}, \mathcal{C}_{44}\}$.

As the environment evolves, new CGSs will be added into LAT as successors of surface CGSs. Surface CGSs whose immediate successors (consistent or inconsistent global states) are not all discovered could have new immediate successor CGSs. However, CGSs whose immediate successors are all discovered could not. For example, in Fig. 1, not all immediate successors of $\{\mathcal{C}_{24}, \mathcal{C}_{34}, \mathcal{C}_{44}\}$ are discovered but all that of $\mathcal{C}_{43}$ are discovered. Thus, $\{\mathcal{C}_{24}, \mathcal{C}_{34}, \mathcal{C}_{44}\}$ could have new immediate successor CGSs but $\mathcal{C}_{43}$ could not. We define the surface CGSs which could have new immediate successors as active CGSs. All the active CGSs in the surface form the active surface.

Definition 2 (Active Surface). *Given the observed lattice LAT and the maximal global state $\mathcal{G}_{max}$, the active surface is defined as follows:*

$$Act(LAT) = \{\mathcal{C} | \mathcal{C} \in Surf(LAT), \exists k, \mathcal{C}[k] = \mathcal{G}_{max}[k]\},$$

where

$$\mathcal{G}_{max} = (s^{(1)}, s^{(2)}, \ldots, s^{(n)}), \forall k, 1 \leq k \leq n,$$
$$s^{(k)} \text{ is the latest local state in } Que^{(k)}.$$

The intuition behind this definition is that, active surface CGSs reach the maximal local state on at least one dimension. This means that in at least one dimension, the successor CGS is yet to come in the future. We are concerned with the active surface, because LAT grows from it. Formally,

Theorem 3. *Given the observed lattice LAT, when a local state $s_j^{(k)}$ arrives, let $Set_{\mathcal{C}}$ denote the new CGSs, i.e., $Set_{\mathcal{C}} = \{\mathcal{C} | \mathcal{C}[k] = s_j^{(k)}, \forall i \neq k, \mathcal{C}[i] \in Que^{(i)}\}$, then*

$$\forall \mathcal{C} : \mathcal{C} \in Set_{\mathcal{C}} :: \exists \mathcal{C}' \in Act(LAT), \mathcal{C}' \rightsquigarrow \mathcal{C}.$$

Proof. To each new CGS $\mathcal{C}$, there exists a CGS sequence $\mathcal{S}(\mathcal{C}_0, \mathcal{C})$. According to the definition of CGS sequences in Section 2.1.2, $\exists \mathcal{C}' \in \mathcal{S}(\mathcal{C}_0, \mathcal{C})$, $\mathcal{C}'[k] = \mathcal{G}_{max}[k]$ (the old $\mathcal{G}_{max}$). $\mathcal{C}'$ was an active CGS. Thus, all new CGSs can grow from the active surface. □

This theorem ensures that keeping only the active surface is sufficient for the growth of LAT. Furthermore, the active surface of the continuously growing lattice always exists, i.e., $Act(LAT) \neq \emptyset$.

To capture the on-going temporal evolution of the environment, we are concerned with the active-surface-induced CGS sequences, that is, CGS sequences that originate from the initial CGS and currently end at active surface CGSs. The reason is that these CGS sequences capture all possible temporal evolutions of the environment state. For example, in Fig. 1, all possible temporal evolutions of the environment state are CGS sequences starting from $\mathcal{C}_0$ and currently ending at CGSs in $\{\mathcal{C}_{24}, \mathcal{C}_{34}, \mathcal{C}_{44}\}$. Formally,

Definition 4 (Evolution). *Given the observed lattice LAT, all possible temporal evolutions of the environment state currently observed are defined as follows:*

$$Evol(LAT) = \{\mathcal{S}(\mathcal{C}_0, \mathcal{C}_i) | \mathcal{C}_i \in Act(LAT)\}$$

3.2 Trace of Environment State Evolution

We regard all Boolean predicates of our concern that are defined on CGSs as an alphabet. Then, the trace of environment state evolution is defined as the formal language obtained by labeling active-surface-induced CGS sequences with letters in the alphabet.

3.2.1 Labeling of CGSs

We specify *CGS predicates* over CGSs to delineate static properties concerning specific snapshot of the environment. CGS predicates consist of local predicates (defined on

constituent local states) connected by binary logic connectors $\wedge, \vee$, and $\Rightarrow$ (imply).

The detection of CGS predicates can be viewed as the labeling of CGSs with letters from an alphabet Σ (i.e., all predefined CGS predicates). This can be denoted by the labeling function $\lambda \subseteq LAT \times 2^{\Sigma}$ (2^{Σ} is the power set of Σ). Fig. 1 is an exemplar labeling with $\Sigma = \{a, b, c\}$. Each CGS in the lattice is labeled with the CGS predicates it satisfies. Please refer to our previous work for more details [17], [20], [21].

3.2.2 Labeling of Active-Surface-Induced CGS Sequences

Based on the labeling $\lambda(\mathcal{C})$ for CGSs, we can further define the labeling $\bar{\lambda}(\mathcal{S}(\mathcal{C}_i, \mathcal{C}_j))$ for CGS sequences. Informally, the labeling of $\mathcal{S}(\mathcal{C}_i, \mathcal{C}_j)$ is the set of all words, where each letter in the word is the labeling of a constituent CGS, i.e.,

$$\bar{\lambda}(\mathcal{S}(\mathcal{C}_i, \mathcal{C}_j)) = \{w | w = w_i w_{i+1} \cdots w_j, \\ \forall k : i \leq k \leq j :: w_k \in \lambda(\mathcal{C}_k)\},$$

Based on the labeling for CGS sequences, we can define the formal language $\mathcal{L}(LAT)$, which is viewed as the trace of environment state evolution. Specifically, we are concerned with active-surface-induced CGS sequences $Evol(LAT)$. Thus, $\mathcal{L}(LAT)$ is the set of words, where each word belongs to the labeling of some active-surface-induced CGS sequence. Formally,

Definition 5 (Trace). *Given the observed lattice LAT, the trace of environment state evolution is defined as*

$$\mathcal{L}(LAT) = \bigcup_{\mathcal{S}(\mathcal{C}_0, \mathcal{C}_i) \in Evol(LAT)} \bar{\lambda}(\mathcal{S}(\mathcal{C}_0, \mathcal{C}_i)).$$

For example, in Fig. 1, we have that $\mathcal{L}(LAT) = \{ccaaaaa, ccaaaaaa, ccaaaaba, ccaaaaaab, ccaaaabab\}$.

The set of active-surface-induced CGS sequences as well as their labeling is essential in modeling of the environment state evolution. It serves as a basis for specification and detection of dynamic properties.

4 Specification of Dynamic Properties

Informally, to specify a dynamic property is to delineate the application's concerns about the environment state evolution. Based on the notion of active surface as well as the trace $\mathcal{L}(LAT)$, the specification of a dynamic property is to specify a set of words that may appear in $\mathcal{L}(LAT)$. We employ regular expressions for the specification. Formal syntax and semantics of the specification are presented below, and an exemplar specification can be found in Section 6.

4.1 Syntax and Semantics of Specification

In the specification of dynamic properties, we employ the syntax of regular expressions [16]. Specifically, dynamic properties Φ can be specified as

$$\Phi ::= \emptyset | \varepsilon | a | \Phi + \Phi | \Phi \cdot \Phi | \Phi^*,$$

with $\emptyset$ denoting the empty set, ε denoting an empty word, and $a \in \Sigma$. Let $\mathcal{L}(\Phi)$ denote the language corresponding to Φ. Detailed definition of $\mathcal{L}(\Phi)$ can be found in Section 3 of the online supplementary file.

In a pervasive computing scenario, P_{che} observes finite prefixes of the potentially infinite environment state evolution. Then it checks whether the trace (i.e., the formal language) of environment state evolution contains words specified by the application. The application expects that satisfaction of the specification can address its concern about the temporal evolution of the environment state.

Formally, we can define whether the trace of environment state evolution satisfies the specification:

$$\mathcal{S}(\mathcal{C}_0, \mathcal{C}_i) \models \Phi \stackrel{def}{=\!=} \bar{\lambda}(\mathcal{S}(\mathcal{C}_0, \mathcal{C}_i)) \cap \mathcal{L}(\Phi) \neq \emptyset.$$

Though this definition is appropriate for deterministic evolution of synchronous environments, it is not sufficient for asynchronous environments. Specifically, we only know partial order (the happen-before relation) between local states on different processes in an asynchronous environment. The checker process obtains multiple possible environment state evolutions, but it never knows which one is the exact evolution, due to the intrinsic uncertainty of the asynchronous environment. This can be better explained by the lattice of CGSs. While monitoring the environment, the checker process observes a continuously growing lattice that contains multiple active-surface-induced CGS sequences, as shown in Fig. 1. It only knows that the environment state evolution is delineated by one of these sequences, but never knows which one.

According to the discussions above, modal operators $Pos(\cdot)$ and $Def(\cdot)$ are necessary for the specification in asynchronous environments [7]. Informally, $Pos(\Phi)$ means that at least one possible CGS sequence satisfies Φ, while $Def(\Phi)$ means that every possible CGS sequence satisfies Φ. Formally,

Definition 6. *Given the observed lattice LAT and the dynamic property Φ,*

$$LAT \models Pos(\Phi) \stackrel{def}{=\!=} \mathcal{L}(LAT) \cap \mathcal{L}(\Phi) \neq \emptyset,$$
$$LAT \models Def(\Phi) \stackrel{def}{=\!=} \mathcal{L}(LAT) \subseteq \mathcal{L}(\Phi).$$

We say that LAT satisfies $Pos(\Phi)$, iff some labeling of some CGS sequence in $Evol(LAT)$ defines a word in $\mathcal{L}(\Phi)$. We say that LAT satisfies $Def(\Phi)$, iff every labeling of every CGS sequence in $Evol(LAT)$ defines a word in $\mathcal{L}(\Phi)$. Note that the satisfaction here is defined over active surface CGSs, while existing work defines the satisfaction over the maximal CGS [2], [22].

For example, let $\Phi' = (a + b + c)^* aa^* b(a + b + c)^*$. In Fig. 1, $\mathcal{L}(LAT) \cap \mathcal{L}(\Phi') \neq \emptyset$, $\mathcal{L}(LAT) \not\subseteq \mathcal{L}(\Phi')$. We can know that $LAT \models Pos(\Phi')$ and $LAT \not\models Def(\Phi')$.

4.2 Regular Expression and Corresponding DFA

To facilitate the detection of the specified property (as detailed in Section 5), we further present the semantic interpretation based on the *DFA* corresponding to the specified regular expression. As we know, there always exists a DFA which accepts a regular expression [16]. Formally, a DFA $\mathcal{A}(\Phi)$ is a 5-tuple $(\Sigma, \mathcal{Q}, q_0, \mathcal{Q}_F, \delta)$ with a

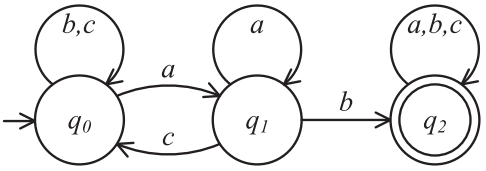

Fig. 2. DFA accepting $\Phi' = (a+b+c)^*aa^*b(a+b+c)^*$.

finite alphabet Σ, a finite set of states $\mathcal{Q}$, an initial state q_0, a set of accepting states $\mathcal{Q}_F$, and a deterministic transition function δ. Fig. 2 illustrates a DFA $\mathcal{A}(\Phi')$ accepting $\Phi' = (a+b+c)^*aa^*b(a+b+c)^*$.

Given the DFA $\mathcal{A}(\Phi)$, let $\mathcal{R}^{\Phi}(Act(LAT))$ denote the set of *reachable states* in $\mathcal{A}(\Phi)$, i.e., states which are reached after processing all the words in $\mathcal{L}(LAT)$. For example, given the DFA $\mathcal{A}(\Phi')$ in Fig. 2 and the lattice LAT in Fig. 1, after injecting all words of $\mathcal{L}(LAT)$ to $\mathcal{A}(\Phi')$, we have $\mathcal{R}^{\Phi'}(Act(LAT)) = \{q_1, q_2\}$. Notice that $\mathcal{R}^{\Phi'}(Act(LAT)) \cap \mathcal{Q}_F \neq \emptyset$ and $\mathcal{R}^{\Phi'}(Act(LAT)) \not\subseteq \mathcal{Q}_F$. Thus, we equivalently get that $LAT \models Pos(\Phi')$ and $LAT \not\models Def(\Phi')$, as we do in Section 4.1. Therefore, we can rewrite Definition 6 as follows.

Definition 7. *Given the observed lattice LAT and the dynamic property Φ,*

$$LAT \models Pos(\Phi) \overset{def}{=\!=} \mathcal{R}^{\Phi}(Act(LAT)) \cap \mathcal{Q}_F \neq \emptyset,$$
$$LAT \models Def(\Phi) \overset{def}{=\!=} \mathcal{R}^{\Phi}(Act(LAT)) \subseteq \mathcal{Q}_F.$$

We compute $\mathcal{R}^{\Phi}(Act(LAT))$ by computing reachable states $\mathcal{R}^{\Phi}(\mathcal{C})$ for each CGS of $Act(LAT)$ [2], i.e.,

$$\mathcal{R}^{\Phi}(Act(LAT)) = \bigcup_{\mathcal{C} \in Act(LAT)} \mathcal{R}^{\Phi}(\mathcal{C}).$$

Therefore, we can rewrite Definition 7 as follows.

Definition 8. *Given the observed lattice LAT and the dynamic property Φ,*

$$LAT \models Pos(\Phi) \overset{def}{=\!=} \exists \mathcal{C} \in Act(LAT), \mathcal{R}^{\Phi}(\mathcal{C}) \cap \mathcal{Q}_F \neq \emptyset,$$
$$LAT \models Def(\Phi) \overset{def}{=\!=} \forall \mathcal{C} \in Act(LAT), \mathcal{R}^{\Phi}(\mathcal{C}) \subseteq \mathcal{Q}_F.$$

Thus, the detection of dynamic properties is reduced to computing the reachable states of active surface CGSs on the DFA corresponding to the specification. This computation is detailed in Section 5.1.3.

5 Detection of the Specified Dynamic Property

In detection of the specified property, we first need to achieve runtime maintenance of the active surface. Then, the detection is reduced to the online recognition of specified regular expression. We first discuss the design rationale. Then we present the detailed design of the *SurfMaint* algorithm, which achieves runtime *maint*enance of the active *surf*ace, and further enables runtime detection of dynamic properties.

5.1 Design Rationale

Detection of dynamic properties is to check whether the trace of environment state evolution can be accepted by the DFA corresponding to the specified property. More importantly, we only maintain the active surface and discard the "old" CGSs of the lattice. We first transform the specified property to its corresponding DFA. Then the active surface is maintained at runtime, and the reachable states of active surface CGSs are computed at runtime based on the discussions in Section 4.2.

5.1.1 Construction of the DFA of the Specification

To facilitate effective detection of the specified property, we transform the regular expression to its corresponding DFA. We employ the standard transformation algorithm from regular expression to NFA and then to DFA [16], and implement it in MIPA [1].

Before the transformation, we first need to revise the specified regular expression. Specifically, runtime monitoring of the environment is to find whether the specified property occurs, i.e., to find whether any word in $\mathcal{L}(\Phi)$ appears as a subsequence of some word in $\mathcal{L}(LAT)$. We extend the original regular expression Φ with prefix and suffix Σ^*, i.e.,

$$\Phi' = \Sigma^*\Phi\Sigma^*.$$

Then our problem essentially boils down to recognition of $\mathcal{L}(\Phi')$ in $\mathcal{L}(LAT)$ [27]. For example, let $\Phi = aa^*b$ be the regular expression specified over $\Sigma = \{a, b, c\}$. The detection of the occurrence of "aa^*b" boils down to the recognition of $\Phi' = (a+b+c)^*aa^*b(a+b+c)^*$. Given the labeled lattice in Fig. 1, we can know that $\mathcal{L}(LAT) \cap \mathcal{L}(\Phi') \neq \emptyset$, i.e., $LAT \models Pos(\Phi')$. Thus, dynamic property $Pos(aa^*b)$ holds.

5.1.2 Runtime Maintenance of the Active Surface

P_{che} collects contextual data from nonchecker processes and then maintains the active surface at runtime. Local states with logical timestamps are sent to P_{che}. P_{che} combines local states to obtain CGSs. Rather than maintaining the whole lattice (as in our previous work [17]), we only maintain the active surface incrementally at runtime. Specifically, when a new local state $s_i^{(k)}$ from $P^{(k)}$ arrives, the newly obtained CGSs which contain $s_i^{(k)}$ are successors of current active surface CGSs, as shown in Theorem 3. These new CGSs replace their predecessor CGSs in the current active surface and the active surface is updated incrementally. With the notion of active surface, runtime evolution of the lattice can be viewed as discarding the "old" active surface and obtain the "new" active surface incrementally at runtime. Based on the active surface, we can further detect whether each new active CGS satisfies predefined CGS predicates, which achieves the labeling of CGSs.

Runtime maintenance of the active surface can be illustrated by the example in Fig. 1. Assume that the current $Que^{(1)} = \{s_0^{(1)}, s_1^{(1)}, s_2^{(1)}, s_3^{(1)}, s_4^{(1)}, s_5^{(1)}\}$ and $Que^{(2)} = \{s_0^{(2)}, s_1^{(2)}, s_2^{(2)}, s_3^{(2)}\}$. Then, the current active surface is $\{\mathcal{C}_{23}, \mathcal{C}_{33}, \mathcal{C}_{43}\}$. When local state $s_4^{(2)}$ arrives, we combine the active CGSs with $s_4^{(2)}$ to obtain a set of new CGSs

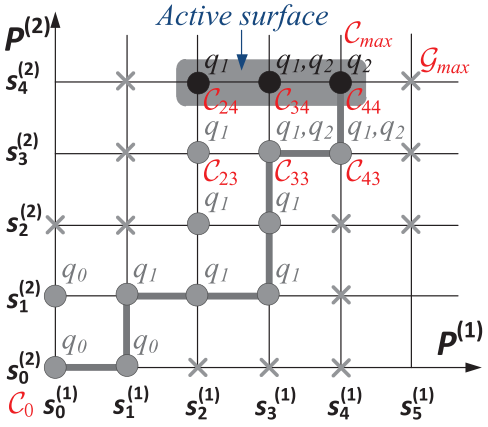

Fig. 3. Lattice of CGSs with reachable states.

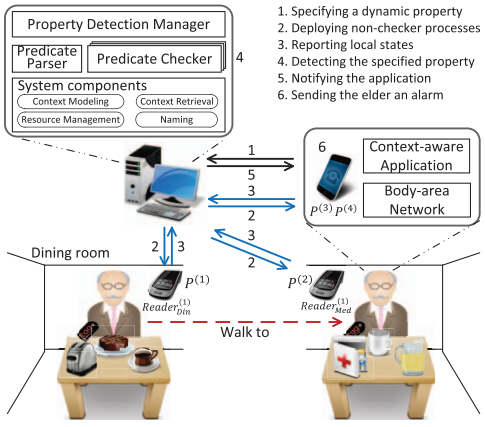

Fig. 4. An elderly care scenario.

$\{\mathcal{C}_{24},\mathcal{C}_{34},\mathcal{C}_{44}\}$. $\{\mathcal{C}_{24},\mathcal{C}_{34},\mathcal{C}_{44}\}$ form the latest active surface, as indicated by the gray rectangle in Fig. 1. CGSs not in the latest active surface are deleted, for example, the gray CGSs in Fig. 1. Newly updated CGSs in the latest active surface are labeled with CGS predicates they satisfy, as shown in Fig. 1.

As we can see, maintaining active surface but not the whole lattice is incremental, and is sufficient for the construction of new active surfaces. Moreover, in the following Section 5.1.3, we will show that it is also sufficient for the computation of reachable states of CGSs in the new active surface.

5.1.3 Computation of Reachable States on $\mathcal{A}(\Phi')$

As $Act(LAT)$ is updated at runtime, reachable states of the remaining CGSs in the active surface will not change. We only need to compute the reachable states of newly obtained CGSs in the active surface. Then, the satisfaction relation (Definition 8) can be checked.

Based on the DFA $\mathcal{A}(\Phi')$, we compute $\mathcal{R}^{\Phi'}(\mathcal{C})$ inductively [2]. First, we set the reachable states of the initial CGS $\mathcal{C}_0$ with $\{q_0\}$ (the initial state of $\mathcal{A}(\Phi')$):

$$\mathcal{R}^{\Phi'}(\mathcal{C}_0) = \{q_0\}.$$

When a new local state $s_i^{(k)}$ arrives, P_{che} produces a set of new CGSs with $\mathcal{C}[k] = s_i^{(k)}$. We first label these CGSs with the CGS predicates they satisfy. For each new CGS $\mathcal{C}$, we can get a set of CGSs that precede ("$\prec$") $\mathcal{C}$ from the "old" and "new" active surfaces, which is denoted as $prec(\mathcal{C})$. Let $\mathcal{R}^{\Phi'}_{prec(\mathcal{C})}$ be the set of reachable states of $\mathcal{A}(\Phi')$ after processing all words labeled on CGS sequences ending with CGSs $\mathcal{C}_j$ in $prec(\mathcal{C})$, i.e.,

$$\mathcal{R}^{\Phi'}_{prec(\mathcal{C})} = \bigcup_{\mathcal{C}_j \in prec(\mathcal{C})} \mathcal{R}^{\Phi'}(\mathcal{C}_j).$$

Thus, by induction we have

$$\mathcal{R}^{\Phi'}(\mathcal{C}) = \bigcup_{q \in \mathcal{R}^{\Phi'}_{prec(\mathcal{C})}, a \in \lambda(\mathcal{C})} \delta(q,a).$$

In this way, the reachable states of new active surface CGSs can be computed incrementally at runtime, as the active surface persistently evolves. Maintaining active surface but not the whole lattice is sufficient for the computation of reachable states of CGSs in the new active surface, as mentioned in Section 5.1.2.

For example, given the DFA $\mathcal{A}(\Phi')$ in Fig. 2 and the lattice LAT in Fig. 1, reachable states of $\{\mathcal{C}_{24},\mathcal{C}_{34},\mathcal{C}_{44}\}$ are shown in Fig. 3, and $\mathcal{R}^{\Phi'}(Act(LAT)) = \{q_1, q_2\}$.

5.2 Design of the SurfMaint Algorithm

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

- Nonchecker processes detect satisfaction of local predicates. If there are message exchanges among nonchecker processes, the happen-before relation can be recorded by piggybacking the vector clock on the messages. Otherwise, SurfMaint lets the nonchecker processes proactively send messages among each other, to establish the happen-before relation. Local states with vector clock timestamps are sent to the checker process.
- On the checker process side, SurfMaint keeps listening to each nonchecker process, maintains the active surface, and detects the specified property at runtime. Both the maintenance of the active surface and the detection of the specified property are incremental.

Detailed design and pseudocodes of the SurfMaint algorithm are presented in Section 4 of the online supplementary file.

6 Case Study

In this section, we conduct a case study to demonstrate how the PDAC framework supports specification and detection of dynamic properties in asynchronous pervasive computing scenarios. We investigate an elderly care scenario, in which an application is persistently aware of the elder's status. As shown in Fig. 4, when the elder tries to take medicine right after dinner, the application may remind him that the medicine should be taken on an empty stomach.

In our example, two RFID readers $Reader_{Din}^{(1)}$ and $Reader_{Med}^{(2)}$ are deployed to detect whether the elder is in the dining room or near the medicine box, respectively. The elder carries a smart phone, which serves as the sink node of the body-area network. The smart phone can, thus, detect the elder's activities (e.g., "eating" and "pouring water into the glass").

Note that the devices do not necessarily have global clocks or shared memory. They suffer from message delay of wireless communications. Moreover, they may postpone the dissemination of context data to save battery power. Thus, the PDAC framework is used to guide the design, implementation, and deployment of the application in this scenario. Each element of PDAC is addressed in detail below.

6.1 Modeling of the Temporal Evolution of Environment State

Nonchecker processes $P^{(1)}$ and $P^{(2)}$ are deployed on $Reader^{(1)}_{Din}$ and $Reader^{(2)}_{Med}$, respectively, to persistently track the location of the elder. Similarly, $P^{(3)}$ and $P^{(4)}$ are deployed on the mobile phone to track the activity of the elder ("eating" or "pouring water"). Thus, the potentially infinite sequences of local states produced by nonchecker processes are "$s^{(k)}_0, s^{(k)}_1, \cdots$" $(1 \leq k \leq 4)$, where in each local state the elder is in some location or conducting some activity. A checker process P_{che} is deployed on a server at home, which is of great computing power and abundant in electric energy.

Nonchecker processes collect local states with logical timestamps, and send them to P_{che}. P_{che} then maintains the active surface at runtime.

6.2 Specification of Dynamic Properties

In our scenario, the application specifies the property "the elder has been having meals, and then he tries to take medicine." To prevent this undesired activity, the application needs to remind the elder even when it only possibly happen. Under the PDAC framework, we first decompose this informal and high-level property to identify local predicates. Then we combine these local predicates and obtain the formal property.

Specifically, according to the property, the application is concerned about the following local predicates:

- $LP^{(1)} =$ "$Reader^{(1)}_{Din}$ *detects the elder.*"
- $LP^{(2)} =$ "$Reader^{(2)}_{Med}$ *detects the elder.*"
- $LP^{(3)} =$ "*Action of the elder is eating.*"
- $LP^{(4)} =$ "*Action of the elder is pouring water.*"

Based on these local predicates, the activity that the elder is having dinner is delineated as "the elder is eating and its location is the dining room." The activity that the elder tries to take medicine is interpreted as "the elder is near the medicine box and is pouring water to the glass." Thus, we can obtain two CGS predicates, which form the alphabet Σ:

- $a = LP^{(1)} \wedge LP^{(3)}$.
- $b = LP^{(2)} \wedge LP^{(4)}$.

Based on these CGS predicates, the property can be delineated by the regular expression $\Phi = aa^*b$ under the PDAC framework. According to the discussions above, the application is to adopt the modal operator $Pos(\cdot)$. Thus, the formal specification to P_{che} is

- $Pos(\Phi) = Pos(aa^*b)$.

6.3 Detection of the Specified Dynamic Property

Upon receiving new local states, P_{che} constructs new CGSs, labels each CGS with the CGS predicates it satisfies, and maintains the active surface at runtime. To ensure that each CGS is labeled with at least one letter, we add a special letter $c = \neg a \wedge \neg b$ to the alphabet. Thus, we have $\Sigma = \{a, b, c\}$.

Upon receiving the property $Pos(aa^*b)$, P_{che} extends $\Phi = aa^*b$ to $\Phi' = (a+b+c)^*aa^*b(a+b+c)^*$ and gets the DFA $\mathcal{A}(\Phi')$ of Fig. 2. Based on $\mathcal{A}(\Phi')$ and the labeling of CGSs, reachable states of each new CGS in $Act(LAT)$ can be obtained. $Pos(\Phi')$ can be detected according to Definition 8. When $Pos(\Phi')$ is true, the application will send the elder an alarm.

7 Performance Measurements

The SurfMaint algorithm is implemented over the open-source context-aware middleware MIPA we developed [1], [40], and the elderly care scenario is simulated. Please refer to Section 5 of the online supplementary file for detailed discussions on the experiment configurations and evaluation results.

Based on the experimental evaluation, we show that the SurfMaint algorithm is capable of detecting the temporal evolution of asynchronous pervasive computing environments. In particular: 1) It can tolerate a reasonable degree of asynchrony (in terms of update interval and message delay), and achieve accurate detection of dynamic properties; 2) It achieves quick response and cost-effectiveness to the temporal evolution of the environment state. Meanwhile, it still suffers to certain extent from the issue of scalability.

8 Related Work

Our proposed PDAC framework can be posed against three areas of related work: context-aware computing, detection of global predicates over asynchronous computations, and traditional model checking with temporal logics.

As for context-aware computing, various schemes have been proposed for detection of contextual properties in the literature. However, it is implicitly assumed in the existing work that the contexts being checked belong to the same snapshot of time, which makes such schemes do not work in asynchronous pervasive computing environments [8], [20], [32], [18].

In detection of global predicates over asynchronous computations, most researchers focus on specific classes of predicates [6], [13], [18], [20], [2], [21], [4], [29], [34]. In our previous work [18], [20], global predicates are detected based on the concurrency among contextual activities. However, they mainly focus on predicates over given snapshot and cannot cope with the temporal evolution of dynamic environments. Hua et al. [17] propose algorithms for runtime construction of the lattice. They do not discuss the active surface, the online maintenance of the active surface, and the detection of dynamic properties.

The PDAC framework is inspired by the detection of dynamic properties in debugging of distributed programs [2], [11], [22]. In these work, the semantics is defined on the CGS sequences spanning from the initial CGS to the final CGS. This is mainly because the distributed program under debugging will finally stop, and it can eventually arrive at a final CGS. However, the semantics is not suitable for

runtime persistent monitoring of the asynchronous pervasive computing environment. As discussed in Section 2.2, when we persistently monitor the asynchronous environment, the lattice continuously grows and will not arrive at a final CGS. To a currently observed lattice, we need to detect properties over all possible CGS sequences that may grow in the future, delineating the temporal evolution of the environment state. We propose the notion of active surface to characterize the environment state evolution at runtime. Thus, the semantics should be defined over active-surface-induced CGS sequences because all possible CGS sequences which may grow in the future currently end with active surface CGSs. We also propose an algorithm for runtime maintenance of the active surface instead of maintaining the whole lattice, which reduces a large amount of space cost.

Our approach shares many similarities with traditional model checking with temporal logics. We use the same formal verification framework (consisting of modeling, specification, and detection). However, there are essential differences [5]. Model checking mainly focuses on verification of a given system at design time, whereas ours mainly focuses on runtime observation of a system in operation. We further compare regular expressions with LTL and dynamic properties with CTL on finite trace of environment state evolution, respectively. The expressive power of regular expressions is larger than that of LTL [36]. A corresponding example can be found in Section 2.2.2. The expressive power of dynamic properties and CTL are overlapping but not contained by each other.

A more detailed discussion on the related work is presented in Section 6 of the online supplementary file.

9 Conclusion and Future Work

In this paper, we study how to enable formal specification and runtime detection of dynamic properties in asynchronous pervasive computing environments. Toward this objective, the PDAC framework is proposed, which consists of three essential parts: 1) modeling of the temporal evolution of the environment state; 2) specification of dynamic properties; and 3) detection of the specified dynamic property.

Currently, the PDAC framework only initiates discussions on one potential approach to enabling specification and detection of dynamic properties in asynchronous pervasive computing environments. Many issues still lack further discussions. In our future work, we need to investigate how to further reduce the space cost for runtime maintenance of the lattice. We also need to investigate whether PDAC can integrate other existing property detection algorithms. Various tools, for example, a GUI for specification of different contextual properties, still need to be implemented to apply the PDAC framework in real context-aware computing scenarios. A more comprehensive and realistic experimental evaluation is also necessary.

Acknowledgments

The authors are grateful to the anonymous reviewers. This work is supported by the National Natural Science Foundation of China (Nos. 61272047, 61021062) and the National 973 Program of China (2009CB320702). Yu Huang is the corresponding author.

References

[1] MIPA - Middleware Infrastructure for Predicate Detection in Asynchronous Environments, http://mipa.googlecode.com, 2013.

[2] O. Babaoğlu, E. Fromentin, and M. Raynal, "A Unified Framework for the Specification and Run-Time Detection of Dynamic Properties in Distributed Computations," *J. Systems and Software,* vol. 33, no. 3, pp. 287-298, 1996.

[3] O. Babaoğlu and K. Marzullo, *Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms.* ACM Press/ Addison-Wesley Publishing Co., 1993.

[4] O. Babaoğlu and M. Raynal, "Specification and Verification of Dynamic Properties in Distributed Computations," *J. Parallel Distributed Computing,* vol. 28, no. 2, pp. 173-185, 1995.

[5] A. Bauer, M. Leucker, and C. Schallhart, "Runtime Verification for Ltl and Tltl," *ACM Trans. Software Eng. and Methodology,* vol. 20, pp. 14:1-14:64, Sept. 2011.

[6] K.M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Trans. Computer Systems,* vol. 3, pp. 63-75, Feb. 1985.

[7] R. Cooper and K. Marzullo, "Consistent Detection of Global Predicates," *Proc. ACM/ONR Workshop Parallel and Distributed Debugging,* pp. 167-174, 1991.

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

[9] G. Dumais and H. Li, "Distributed Predicate Detection in Series-Parallel Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 13, no. 4, pp. 373-387, Apr. 2002.

[10] C.J. Fidge, "Partial Orders for Parallel Debugging," *Proc. ACM SIGPLAN and SIGOPS Workshop Parallel and Distributed Debugging,* pp. 183-194, May 1988.

[11] E. Fromentin, M. Raynal, V.K. Garg, and A. Tomlinson, "On the Fly Testing of Regular Patterns in Distributed Computations," *Int'l Conf. Parallel Processing,* vol. 2, pp. 73-76, 1994.

[12] V. Garg and B. Waldecker, "Detection of Weak Unstable Paredicates in Distributed Programs," *IEEE Trans. Parallel and Distributed Systems,* vol. 5, no. 3, pp. 299-307, Mar. 1994.

[13] 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.

[14] 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.

[15] K. Henricksen and J. Indulska, "Developing Context-Aware Pervasive Computing Applications: Models and Approach," *Pervasive and Mobile Computing,* vol. 2, no. 1, pp. 37-64, 2006.

[16] J.E. Hopcroft, R. Motwani, and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation,* second ed. Addison-Wesley, 2000.

[17] 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,* pp. 307-321, 2010.

[18] 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.

[19] Y. Huang, X. Ma, X. Tao, J. Cao, and J. Lu, "A Probabilistic Approach to Consistency Checking for Pervasive Context," *Proc. IEEE/IFIP Intl. Conf. Embedded and Ubiquitous Computing,* pp. 387-393, Dec. 2008.

[20] Y. Huang, Y. Yang, J. Cao, X. Ma, X. Tao, and J. Lu, "Runtime Detection of the Concurrency Property in Asynchronous Pervasive Computing Environments," *IEEE Trans. Parallel and Distributed Systems,* vol. 23, no. 4, pp. 744-750, Apr. 2012.

[21] 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,* pp. 75-82, 2010.

[22] C. Jard, G.-V. Jourdan, T. Jeron, and J.-X. Rampon, "A General Approach to Trace-Checking in Distributed Computing Systems," *Proc. Int'l Conf. Distributed Computing Systems,* pp. 396-403, June 1994.

[23] 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.

[24] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM,* vol. 21, no. 7, pp. 558-565, 1978.
[25] H. Lu, W.K. Chan, and T.H. Tse, "Testing Context-Aware Middleware-Centric Programs: A Data Flow Approach and an RFID-Based Experimentation," *Proc. ACM SIGSOFT Int'l Symp. Foundations of Software Eng. (FSE '06),* pp. 242-252, 2006.
[26] J. Lu, X. Ma, X. Tao, C. Cao, Y. Huang, and P. Yu, "On Environment-Driven Software Model for Internetware," *Science in China Series F: Information Sciences,* vol. 51, no. 6, pp. 683-721, 2008.
[27] A. Majumder, R. Rastogi, and S. Vanama, "Scalable Regular Expression Matching on Data Streams," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 161-172, 2008.
[28] F. Mattern, "Virtual Time and Global States of Distributed Systems," *Proc. Int'l Workshop Parallel and Distributed Algorithms,* pp. 215-226, 1989.
[29] N. Mittal, A. Sen, and V. Garg, "Solving Computation Slicing Using Predicate Detection," *IEEE Trans. Parallel and Distributed Systems,* vol. 18, no. 12, pp. 1700-1713, Dec. 2007.
[30] A. Ranganathan and R.H. Campbell, "An Infrastructure for Context-Awareness Based on First Order Logic," *Personal Ubiquitous Computing,* vol. 7, no. 6, pp. 353-364, 2003.
[31] M. Roman, C. Hess, R. Cerqueira, A. Ranganathan, R.H. Campbell, and K. Nahrstedt, "A Middleware Infrastructure for Active Spaces," *IEEE Pervasive Computing,* vol. 1, no. 4, pp. 74-83, Oct.-Dec. 2002.
[32] M. Sama, S. Elbaum, F. Raimondi, D.S. Rosenblum, and Z. Wang, "Context-Aware Adaptive Applications: Fault Patterns and Their Automated Identification," *IEEE Trans. Software Eng.,* vol. 36, no. 5, pp. 644-661, Sept./Oct. 2010.
[33] R. Schwarz and F. Mattern, "Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail," *Distributed Computing,* vol. 7, no. 3, pp. 149-174, 1994.
[34] A. Sen and V. Garg, "Formal Verification of Simulation Traces Using Computation Slicing," *IEEE Trans. Computers,* vol. 56, no. 4 pp. 511-527, Apr. 2007.
[35] K. Sen and G. Roşu, "Generating Optimal Monitors for Extended Regular Expressions," *Proc. Third Int'l Workshop Runtime Verification (RV '03),* pp. 162-181, 2003.
[36] J. Strejcek, "Linear Temporal Logic: Expressiveness and Model Checking," PhD thesis, Faculty of Informatics, Masaryk Univ. in Brno, 2004.
[37] 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.
[38] 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.
[39] F. Yang, J. Lu, and H. Mei, "Technical Framework for Internetware: An Architecture Centric Approach," *Science in China Series F: Information Sciences,* vol 51, no. 6, pp. 610-622, 2008.
[40] J. Yu, Y. Huang, J. Cao, and X. Tao, "Middleware Support for Context-Awareness in Asynchronous Pervasive Computing Environments," *Proc. IEEE/IFIP Int'l Conf. Embedded and Ubiquitous Computing,* pp. 136-143, 2010.

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. He is a student member of the IEEE.

Yu Huang received the BSc and PhD degrees in computer science from the University of Science and Technology of China, Hefei, 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.

Jiannong Cao received the BSc degree in computer science from Nanjing University, China, in 1982, and the MSc and PhD degrees in computer science from Washington State University, Pullman, Washington, in 1986 and 1990, respectively. He is currently a chair professor in the Department of Computing at Hong Kong Polytechnic University, Hung Hom. He is also the director of the Internet and Mobile Computing Lab in the department. Before joining Hong Kong Polytechnic University, he was a faculty member in the Department 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 300 technical papers in the above areas. His recent research has focused on mobile and pervasive computing systems, developing testbed, 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 the IEEE, including Computer Society and the IEEE Communication Society, a senior member of the China Computer Federation, 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.

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. 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.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.

Supplementary File of "Formal Specification and Runtime Detection of Dynamic Properties in Asynchronous Pervasive Computing Environments"

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

1 Introduction

In our paper "Formal Specification and Runtime Detection of Dynamic Properties in Asynchronous Pervasive Computing Environments", we propose the Property Detection for Asynchronous Context (PDAC) framework to support formal specification and runtime detection of dynamic properties in asynchronous pervasive computing environments. Three essential parts of the framework are discussed: modeling of the temporal evolution of environment state, specification of dynamic properties, and detection of the specified dynamic property by the proposed *SurfMaint* algorithm.

In this supplementary file, we further provide more details about how we address these three essential parts. Specifically, Section 2 gives the definition of the happen-before relation and logical vector clock. Section 3 gives the definition of the languages corresponding to regular expressions. Section 4 and 5 provide the detailed design and performance measurements of the SurfMaint algorithm, respectively. Section 6 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.

- *Corresponding author: Yu Huang, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China, 210046. E-mail: yuhuang@nju.edu.cn.*
- *Yiling Yang, Xiaoxing Ma, and Jian Lu are with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China, 210046. E-mail: csylyang@gmail.com, {xxm, lj}@nju.edu.cn.*
- *Jiannong Cao is with the Hong Kong Polytechnic University. E-mail: csjcao@comp.polyu.edu.hk.*

2 Definition of the Happen-Before Relation and Logical Vector Clock

We re-interpret the notion of time based on Lamport's definition of the *happen-before* relation (denoted by '$\rightarrow$') resulting from message causality [19], as well as the coding of this happen-before relation by the logical vector clock [21].

Specifically, while monitoring specific regions or aspects of the environment, each non-checker process $P^{(k)}$ generates its (potentially infinite) trace of *local states* connected by *contextual events*: "$e_0^{(k)}, s_0^{(k)}, e_1^{(k)}, s_1^{(k)}, e_2^{(k)}, s_2^{(k)}, \cdots$". Contextual events may be local, e.g. update of context data, or global, e.g. sending/receiving messages. For two contextual events e_1 and e_2, we have $e_1 \rightarrow e_2$ iff:

- Events e_1 and e_2 are on the trace of the same non-checker process and e_1 is generated before e_2, or
- Events e_1 and e_2 are on traces of 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 and s_2, $s_1 \rightarrow s_2$ iff the ending of s_1 happen-before (or coincides with) the beginning of s_2 (note that the beginning and ending of a local state are both contextual events).

We use the logical vector clock [21] to depict the happen-before relation among local states. Specifically, each $P^{(j)}$ keeps its vector clock timestamp $VC^{(j)}$:

- $VC^{(j)}[i](i \neq j)$ is the ID of the latest event from $P^{(i)}$ (the ID of event $e_k^{(j)}$ is k), which has a causal relation to $P^{(j)}$;
- $VC^{(j)}[j]$ is the ID of the next event $P^{(j)}$ will produce.

3 Definition of the Languages Corresponding to Regular Expressions

In the specification of dynamic properties, we employ the syntax of regular expressions [13], [4]. Specifically,

dynamic properties Φ can be specified as:

$$\Phi ::= \varnothing|\varepsilon|a|\Phi + \Phi|\Phi \cdot \Phi|\Phi^*,$$

with $\varnothing$ denoting the empty set, ε denoting an empty word, and $a \in \Sigma$.

Let $\mathcal{L}(\Phi)$ denote the language corresponding to Φ, which is defined as follows:

$\mathcal{L}(\varnothing) = \varnothing$, $\mathcal{L}(\varepsilon) = \{\varepsilon\}$
$\mathcal{L}(a) = \{a\}$ (for each $a \in \Sigma$)
$\mathcal{L}(\Phi_1 + \Phi_2) = \mathcal{L}(\Phi_1) \cup \mathcal{L}(\Phi_2)$ $(union)$
$\mathcal{L}(\Phi_1 \cdot \Phi_2) = \mathcal{L}(\Phi_1)\mathcal{L}(\Phi_2)$ $(composition)$
$\mathcal{L}(\Phi^*) = \bigcup_{i\in\mathbb{N}} \mathcal{L}(\Phi)^i$ $(iteration)$

Here, Φ^* denotes the Kleene-star [13] or iterator which is recursively interpreted as follows:

$\mathcal{L}(\Phi)^0 = \{\varepsilon\}$
$\mathcal{L}(\Phi)^{i+1} = \{uv|u \in \mathcal{L}(\Phi), v \in \mathcal{L}(\Phi)^i\}$ $(i \in \mathbb{N})$

4 Detailed Design of the SurfMaint Algorithm

In this section, we present the design of the SurfMaint algorithm on both non-checker and checker process sides. Unlike traditional distributed systems in which processes communicate with each other via messages, different processes in the pervasive system may be independent and may not (need to) communicate with each other. In our PDAC framework, we discuss both cases where the processes in the pervasive system may or may not communicate with each other. Specifically, if there are message exchanges among the processes involved in the detection of contextual predicates, the happen-before relation can be inferred by piggybacking the vector clock on the messages, as in traditional distributed systems. If there are no message exchanges among the processes, in order to infer the temporal relation among contextual events on different processes, the processes may need to proactively send messages (control messages) among each other to establish the happen-before relation. The operation of SurfMaint involves three different types of messages:

- *Application message.* Messages among non-checker processes generated by other applications are called application messages.
- *Control message.* Non-checker processes send control messages among each other to establish the happen-before relation required for coping with the asynchrony by logical time, if there are no application messages.
- *Checking message.* The checker process collects local states in checking messages from non-checker processes, maintains the active surface, and detects the specified dynamic property.

4.1 SurfMaint on the Non-checker Process Side

Each $P^{(k)}$ is in charge of collecting contextual local states and checking local predicates. $P^{(k)}$ also maintains a vector clock $VC^{(k)}$. When local predicate turns true, $P^{(k)}$ sends a control message to other processes, to establish the happen-before relation between local activities on different processes despite of the asynchrony. When local predicate turns false, $P^{(k)}$ updates $VC^{(k)}[k]$. When $P^{(k)}$ sends an application message to other processes, it piggybacks the current vector clock $VC^{(k)}$ on the message. When $P^{(k)}$ receives a message (application message or control message) from other processes, it updates the vector clock $VC^{(k)}$. Whenever $P^{(k)}$ proceeds to a new local state, it sends a checking message to P_{che}. Pseudo codes of SurfMaint on $P^{(k)}$ are listed in Algorithm 1.

Algorithm 1: SurfMaint on $P^{(k)}$

```
1  Upon LP^(k) becomes true
2  send control message(VC^(k)) to each P^(j)(j ≠ k);
3  VC^(k)[k]++;
4  Upon LP^(k) becomes false
5  VC^(k)[k]++;
6  Upon Receiving message(VC^(j)) from P^(j)
7  for i = 1 to n do
8      VC^(k)[i] = max{VC^(k)[i], VC^(j)[i]};
9  VC^(k)[k]++;
10 Upon Sending an application message to P^(j)
11 piggyback VC^(k) on the application message;
12 VC^(k)[k]++;
13 Upon P^(k) proceeds to a new local state s_i^(k)
14 send checking message(s_i^(k)) to P_che;
```

Algorithm 2: SurfMaint on P_{che}

```
1  Upon Initialization
2  get property φ = Pos(Φ) or Def(Φ);
3  extend Φ into Φ′; transform Φ′ into DFA A(Φ′);
4  add C_0 to Act(LAT); C_max = G_max = C_0;
5  Upon Receiving checking-msg(s_i^(k))
6  grow_new(Act(LAT), s_i^(k));   /* Algorithm 3 */
7  check(Act(LAT), A(Φ′));        /* Algorithm 4 */
8  prune_old(Act(LAT));           /* Algorithm 5 */
```

4.2 SurfMaint on the Checker Process Side

P_{che} keeps listening to each $P^{(k)}$, maintains the active surface, and detects the specified property at runtime. Both the maintenance of the active surface and the detection of the specified property are incremental, i.e., when new CGSs are constructed, P_{che} checks whether these new CGSs make the specified property true. The entire lattice is never constructed. Pseudo codes of SurfMaint on P_{che} are listed in Algorithm 2.

During the initialization, we first get the property $Pos(\Phi)$ or $Def(\Phi)$, extend Φ to Φ', and obtain the DFA $\mathcal{A}(\Phi')$. When a new local state $s_i^{(k)}$ arrives, we maintain the active surface and check the specified

Algorithm 3: $grow\_new(Act(LAT), s_i^{(k)})$

1 add $s_i^{(k)}$ to $Que^{(k)}$; update $\mathcal{G}_{max}$;
2 combine $\mathcal{C}_{max}$ and $s_i^{(k)}$ to get a global state $\mathcal{G}$;
3 **if** $\mathcal{G}$ *is CGS* **then**
4 connect $\mathcal{C}_{max}$ to $\mathcal{G}$; $grow(\mathcal{G})$;
5 $Act(LAT) = \{\mathcal{C} | \mathcal{C} \in LAT, \exists k, \mathcal{C}[k] = \mathcal{G}_{max}[k]\}$;

subroutine $grow(\mathcal{C})$
1 $prec(\mathcal{C}) = \{\mathcal{C}' | \forall i, \mathcal{C}'[i] \in Que^{(i)}, \mathcal{C}'$ is CGS, $\mathcal{C}' \prec \mathcal{C}\}$;
2 $sub(\mathcal{C}) = \{\mathcal{C}' | \forall i, \mathcal{C}'[i] \in Que^{(i)}, \mathcal{C}'$ is CGS, $\mathcal{C} \prec \mathcal{C}'\}$;
3 **if** $sub(\mathcal{C}) = \emptyset$ **then** $\mathcal{C}_{max} = \mathcal{C}$;
4 **foreach** $\mathcal{C}'$ *in* $prec(\mathcal{C})$ **do**
5 **if** $\mathcal{C}' \notin LAT$ **then**
6 connect $\mathcal{C}'$ to $\mathcal{C}$; $grow(\mathcal{C}')$;
7 **foreach** $\mathcal{C}'$ *in* $sub(\mathcal{C})$ **do**
8 **if** $\mathcal{C}' \notin LAT$ **then**
9 connect $\mathcal{C}$ to $\mathcal{C}'$; $grow(\mathcal{C}')$;

property. The maintenance of active surface consists of adding new active CGSs and discarding old inactive CGSs. Pseudo codes of adding new active CGSs, checking the specified property, and discarding old CGSs are listed in Algorithm 3, 4, and 5, respectively.

- *Grow new active CGSs.* When adding new CGSs, we combine $s_i^{(k)}$ with $\mathcal{C}_{max}$ to obtain a global state $\mathcal{G}$, as shown in line 2 of Algorithm 3. The lattice can grow, iff $\mathcal{G}$ is CGS (Please refer to the proof of Theorem 4.1 in [29].). The growing of new active CGSs is achieved by recursively adding their predecessors and successors, as shown in the subroutine $grow(\mathcal{C})$. Theorem 3 in the main file ensures that the active surface is sufficient for the growth of new CGSs. During the growing process, $\mathcal{C}_{max}$ is also updated in line 3 of the subroutine. The new active surface is then defined in line 5 of Algorithm 3.
- *Check the specified property.* After the growing of new CGSs, the reachable states of new CGSs are computed in the subroutine $compute\_reachable\_states(\mathcal{C})$ of Algorithm 4. The computation of reachable states is achieved by recursively computing reachable states of all the predecessors of a CGS. During the process, each new CGS is labeled with the CGS predicates it satisfies in line 1 of the subroutine. Then, referred to Definition 8 in the main file, the specified dynamic property is detected according to the relation between the reachable states of the active surface CGSs and the accepting states of the DFA $\mathcal{A}(\Phi')$, as shown in line 2-10 of Algorithm 4.
- *Prune old inactive CGSs.* After checking the specified property, inactive CGSs are discarded, as shown in Algorithm 5.

Algorithm 4: $check(Act(LAT), \mathcal{A}(\Phi'))$

1 $compute\_reachable\_states(\mathcal{C}_{max})$;
2 Boolean $flag = true$;
3 $Set_{accept} = \mathcal{A}(\Phi').getAcceptingStates()$;
4 **foreach** $\mathcal{C}$ *in* $Act(LAT)$ **do**
5 $Set_{reach} = \mathcal{C}.getReachableStates()$;
6 **if** $Set_{reach} \cap Set_{accept} \neq \varnothing$ **then**
7 $Pos(\Phi')$ is true; /* $Pos(\Phi')$ return */
8 **if** $Set_{reach} \not\subseteq Set_{accept}$ **then**
9 $flag = false$; /* $Def(\Phi')$ break */
10 **if** $flag = true$ **then** $Def(\Phi')$ is true;

subroutine $compute\_reachable\_states(\mathcal{C})$
1 $label\_CGS\_predicates(\mathcal{C})$;
2 $prec(\mathcal{C}) = \{\mathcal{C}' | \mathcal{C}' \in LAT, \mathcal{C}' \prec \mathcal{C}\}$;
3 **foreach** *CGS* $\mathcal{C}'$ *in* $prec(\mathcal{C})$ **do**
4 **if** $\mathcal{C}'.getReachableStates() = \varnothing$ **then**
5 $compute\_reachable\_states(\mathcal{C}')$;
6 $\mathcal{R}^{\Phi'}_{prec(\mathcal{C})} = \bigcup_{\mathcal{C}' \in prec(\mathcal{C})} \mathcal{R}^{\Phi'}(\mathcal{C}')$;
7 **foreach** *state* q *in* $\mathcal{R}^{\Phi'}_{prec(\mathcal{C})}$ **do**
8 **foreach** *letter* a *in* $\mathcal{C}.getLabeling()$ **do**
9 **if** $\delta(q, a) \notin \mathcal{C}.getReachableStates()$ **then**
10 $\mathcal{C}.addReachableState(\delta(q, a))$;

Algorithm 5: $prune\_old(Act(LAT))$

1 **foreach** *CGS* $\mathcal{C}$ *in* LAT **do**
2 Boolean $flag = false$;
3 **for** i = *1* **to** n **do**
4 **if** $\mathcal{C}[i] = \mathcal{G}_{max}[i]$ **then**
5 $flag = true$; /* $\mathcal{C}$ is active CGS */
6 break;
7 **if** *flag = false* **then**
8 delete $\mathcal{C}$; /* inactive CGSs are deleted */

4.3 Complexity Analysis

First, we discuss the construction of the DFA. Given a regular expression Φ, we can get the corresponding NFA in $O(m)$, where m is the length of Φ. The conversion from NFA to DFA is in $O(m^3 2^m)$. However, in practice, it is common to take $O(m^3 s)$ as a bound, where s is the number of states the DFA contains [13]. Moreover, we find that the number of states of the DFA corresponding to extended regular expression Φ' ($\Phi' = \Sigma^* \Phi \Sigma^*$) is never greater than that of the NFA corresponding to Φ (see Section 2.4.3 in [13]). Thus, the conversion from NFA to DFA is in $O(m^4)$. Thus, the complexity of line 3 of Algorithm 2 is $O(m^4)$. Specifically, in pervasive computing scenarios, m is usually on a small scale.

Second, we discuss the runtime maintenance of the

active surface. Regarding the space for a single CGS as one unit, the worst-case space cost of active surface is $O(np^{n-1})$, where p is the upper bound of number of local states of each non-checker process, and n is the number of non-checker processes. However, the worst-case space cost of lattice is $O(p^n)$. Due to the incremental nature of Algorithm 3 and 5, the worst-case space cost of the new part of active surface in each time of growing is $O(p^{n-1})$ and the released space of the old part in each time of pruning is $O(p^{n-1})$. Furthermore, in most pervasive computing scenarios, the space cost of maintaining active surface is usually much less than that of preserving the whole lattice (see more discussions in Section 5).

Finally, we discuss the detection of the specified property. In Algorithm 4, the time cost of computing reachable states for each CGS is $O(mn)$. In each time of growing, only the new added CGSs of the active surface have to be computed. Thus, the worst-case time cost of Algorithm 4 is $O(mnp^{n-1})$.

5 Performance Measurements

In this section, we conduct experiments to obtain quantitative performance measurements for the SurfMaint algorithm. We first describe the implementation. Then we describe the experiment setup. Finally we discuss the evaluation results.

5.1 Implementation

The detection of dynamic properties assumes the availability of an underlying context-aware middleware [20], [28]. We have implemented the middleware based on one of our research projects - *Middleware Infrastructure for Predicate detection in Asynchronous environments* (MIPA) [1], [30]. The system architecture of MIPA is shown in Fig. 1.

From MIPA's point of view, the application achieves context-awareness by specifying dynamic properties of its interest to MIPA. Checker processes are implemented as third-party services, plugged into MIPA. Non-checker processes are deployed (on ECA in Fig. 1) to manipulate context collecting devices, monitor different regions of the environment, and disseminate context data to MIPA.

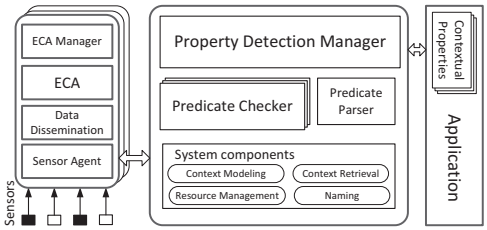

Fig. 1. System architecture of MIPA

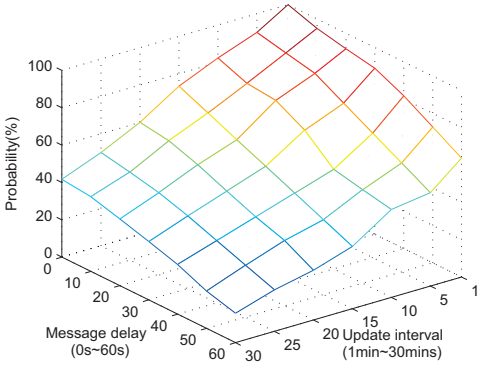

Fig. 2. Asynchrony vs. probability of detection

5.2 Experiment Setup

We simulate the scenario discussed in our case study in the main file. Specifically, sensors collect context data every 1 min. Duration of local contextual activities on non-checker processes follows the Poisson process. The average duration of contextual activities is 25 mins and the interval between contextual activities is 5 mins[1]. Lifetime of the experiment is up to 150 hours.

In the experiments, we study the impact of asynchrony of environments on the *Probability of Detection* of the specified property $Prob_{det}$, the *Response Time* of property detection $Cost_t$, and the *Space Cost* of property detection $Cost_s$. $Prob_{det}$ is calculated as the ratio of $\frac{N_{SurfMaint}}{N_{physical}}$. Here, $N_{SurfMaint}$ denotes the number of times SurfMaint detects the specified dynamic property. $N_{physical}$ denotes the number of times such property holds in the physical world. $Cost_t$ denotes the time from the instant when P_{che} is triggered to the instant when the detection finishes. $Cost_s$ denotes the average size of the active surface when SurfMaint detects the specified dynamic property.

To study the impact of asynchrony on SurfMaint, we tune the interval between the sensors update data to non-checker processes (denoted as update interval) and the average message delay between non-checker processes. We also tune the number of non-checker processes to investigate the performance of SurfMaint.

5.3 Effects of Tuning the Asynchrony

In this experiment, we study how the message delay and update interval affect the performance of SurfMaint. We tune the average message delay from 0 s to 60 s, and the update interval from 1 min to 30 mins.

As shown in Fig. 2, the message delay and update interval both result in monotonic decrease in $Prob_{det}$, mainly due to the increasing uncertainty caused by the asynchrony. When encountered with reasonable asynchrony (average message delay less than 20 s and update interval less than 10 mins), $Prob_{det}$ remains high (around 90%). To a fixed update interval,

1. We increase the frequency of occurrence of contextual activities, in order to collect sufficient amount of experiment data.

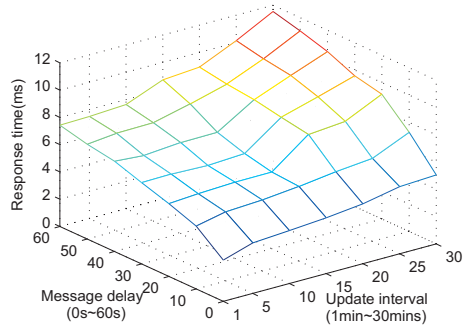

Fig. 3. Asynchrony vs. response time

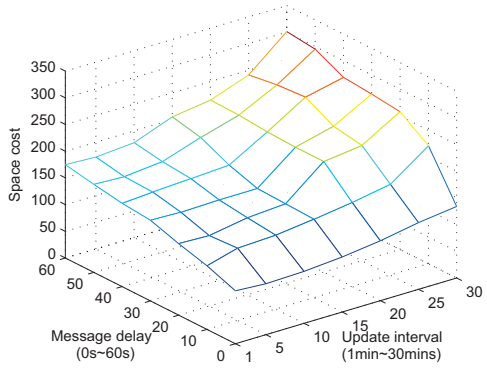

Fig. 4. Asynchrony vs. space cost

when we tune the average message delay (0 s to 60 s), $Prob_{det}$ decreases about 35%. To a fixed average message delay, when we tune the update interval (1 min to 30 mins), $Prob_{det}$ decreases about 55%. The impact of message delay is comparatively less than that of the update interval. It is mainly because that the message delay is usually in a small scale, while the update interval may be increased to a quite large value (usually to save energy of the battery-powered context collecting devices).

As shown in Fig. 3 and 4, the message delay and update interval both result in monotonic slow increase in $Cost_t$ and $Cost_s$. The increases of both $Cost_t$ and $Cost_s$ are mainly due to that the detection faces more CGSs caused by the increasing uncertainty of asynchrony. When we tune the average message delay (0 s to 60 s) and update interval (1 min to 30 mins), $Cost_t$ increases monotonously about 10 ms and remains within 12 ms, and $Cost_s$ increases monotonously about 200 and remains within 350, whereas the size of the whole lattice is around 250,000. $Cost_s$ is much less than the space cost of maintaining the whole lattice. This is in accordance with our analysis in Section 4.3.

5.4 Effects of Tuning the Number of Non-checker Processes

In this experiment, we study how the number of non-checker processes affects the performance of SurfMaint. We fix the average message delay to 0.5 s and update interval to 5 mins. We tune the number of non-checker processes from 2 to 7. The lifetime is 5 hours.

TABLE 1
Tuning the number of non-checker processes

Non-checker process number	$Prob_{det}$ (%)	$Cost_t$ (ms)	$Cost_s$	Size of lattice
2	100	0.81	3	84
3	100	1.11	22	829
4	89	6.07	157	6899
5	82	291.62	2448	165945
6	76	16430.73	47119	3291014
7	60	46242.41	104417	7435693

As shown in Table 1, $Prob_{det}$ decreases as the number of non-checker processes increases, mainly due to the increasing asynchrony caused by the increase of the number of non-checker processes. $Cost_t$ and $Cost_s$ both greatly increase as the number of non-checker processes increases, which is in accordance with the analysis in Section 4.3. $Cost_s$ is much less than the space cost of the whole lattice. SurfMaint reduces a large amount of space cost by runtime and incremental maintenance of the active surface.

6 Additional Discussions on the Related Work

Our proposed PDAC framework can be posed against three areas of related work: context-aware computing, detection of global predicates over asynchronous computations, and traditional model checking with temporal logics.

As for context-aware computing, in [27], properties were modeled by tuples, and property detection was based on comparison among elements in the tuples. In [28], contextual properties were expressed in first-order-logic, and an incremental property detection algorithm was proposed.

In detection of global predicates over asynchronous computations, Cooper et al. [8] investigated the detection of general predicates, which brought combinatorial explosion of the state space. Most researchers focus on specific classes of predicates: *snapshot predicates* and *behavior predicates*. Snapshot predicates include the stable predicates [6], the linear predicates [7], the conjunctive predicates [11], [12], [14], [15], the relational predicates [26], etc. Behavior predicates include regular expression predicates [2], [9], [18] (including the linked predicates [22], the simple sequence predicates [17], [16], the interval-constrained sequence predicates [3], etc.) and temporal logic predicates [10], [23], [24].

Our approach shares many similarities with traditional model checking with temporal logics. We use the same formal verification framework (consisting of modeling, specification, and detection). Specifically, a formal model is used to capture how the system operates. User's concerns are expressed in formal languages. The verification is conducted automatically

by carefully-designed checking algorithms. However, there are essential differences [5]. Model checking mainly focuses on verification of a given system at design time, whereas ours mainly focuses on runtime observation of a system in operation for further runtime adaptations of context-aware applications. Accordingly, in model checking, a precise description of the system is mandatory before actually running the system. In contrast, as an external observer of an already-running system, our approach is applicable to "black box" systems for which no system model is at hand. Model checking deals with infinite traces of all possible executions, whereas our approach deals with observed finite prefixes of potentially infinite traces of one concrete execution. Consequently, the cost of our approach is fairly smaller than that of model checking, which suffers from the so-called state explosion problem.

Interpreting a regular expression over a single CGS sequence corresponds to model checking with LTL. Interpreting a dynamic property over the active-surface-induced CGS sequences corresponds to model checking with CTL. We compare regular expressions with LTL and dynamic properties with CTL on finite trace of environment state evolution, respectively. It has been proved that LTL and star-free regular expressions have the same expressive power [25]. Thus, the expressive power of regular expressions is larger than that of LTL. The universal and existential quantifications can be nested in CTL formulae, while our dynamic properties can have only one modal operator Pos or Def in front. Thus, the expressive power of dynamic properties and CTL are overlapping but not contained by each other.

REFERENCES

[1] MIPA - Middleware Infrastructure for Predicate detection in Asynchronous environments. http://mipa.googlecode.com.

[2] O. Babaoğlu, E. Fromentin, and M. Raynal. A unified framework for the specification and run-time detection of dynamic properties in distributed computations. *J. of Syst. and Softw.*, 33(3):287–298, 1996.

[3] O. Babaoğlu and M. Raynal. Specification and verification of dynamic properties in distributed computations. *J. Parallel Distrib. Comput.*, 28(2):173–185, 1995.

[4] A. Bauer. *Model-based runtime analysis of distributed reactive systems*. PhD thesis, Institut für Informatik, Technische Universität München, 2007.

[5] A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for ltl and tltl. *ACM Trans. Softw. Eng. Methodol.*, 20:14:1–14:64, Sep 2011.

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

[7] C. M. Chase and V. K. Garg. Detection of global predicates: Techniques and their limitations. *Distrib. Comput.*, 11:191–201, 1998.

[8] R. Cooper and K. Marzullo. Consistent detection of global predicates. In *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 167–174, 1991.

[9] E. Fromentin, M. Raynal, V. K. Garg, and A. Tomlinson. On the fly testing of regular patterns in distributed computations. In *Intl. Conf. on Parallel Processing, Vol 2*, pages 73–76, 1994.

[10] V. Garg and N. Mittal. On slicing a distributed computation. In *Intl. Conf. on Distrib. Comput. Syst.*, pages 322–329, Apr 2001.

[11] V. Garg and B. Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Trans. Parallel Distrib. Syst.*, 5:299–307, Mar 1994.

[12] V. K. Garg and B. Waldecker. Detection of strong unstable predicates in distributed programs. *IEEE Trans. Parallel Distrib. Syst.*, 7:1323–1333, Dec 1996.

[13] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation, 2nd Edition*. Addison-Wesley, 2000.

[14] Y. Huang, X. Ma, J. Cao, X. Tao, and J. Lu. Concurrent event detection for asynchronous consistency checking of pervasive context. In *IEEE Intl. Conf. on Pervasive Computing and Communications (PERCOM'09)*, Mar 2009.

[15] Y. Huang, Y. Yang, J. Cao, X. Ma, X. Tao, and J. Lu. Runtime detection of the concurrency property in asynchronous pervasive computing environments. *IEEE Trans. Parallel Distrib. Syst.*, 23(4):744–750, Apr 2012.

[16] Y. Huang, J. Yu, J. Cao, and X. Tao. Detection of behavioral contextual properties in asynchronous pervasive computing environments. *Intl. Conf. on Parallel and Distrib. Syst.*, pages 75–82, 2010.

[17] M. Hurfin, N. Plouzeau, and M. Raynal. Detecting atomic sequences of predicates in distributed computations. In *Proc. of the 1993 ACM/ONR workshop on Parallel and distributed debugging*, pages 32–42, 1993.

[18] C. Jard, G.-V. Jourdan, T. Jeron, and J.-X. Rampon. A general approach to trace-checking in distributed computing systems. In *Intl. Conf. on Distrib. Comput. Syst.*, pages 396–403, Jun 1994.

[19] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[20] H. Lu, W. K. Chan, and T. H. Tse. Testing context-aware middleware-centric programs: a data flow approach and an rfid-based experimentation. In *Proc. ACM SIGSOFT Intl. Symp. on Foundations of Softw. Eng. (FSE'06)*, pages 242–252, 2006.

[21] F. Mattern. Virtual time and global states of distributed systems. In *Proc. Intl. Workshop on Parallel and Distributed Algorithms*, pages 215–226, 1989.

[22] B. Miller and J.-D. Choi. Breakpoints and halting in distributed programs. In *Proc. Intl. Conf. on Distrib. Comput. Syst.*, pages 316–323, Jun 1988.

[23] N. Mittal, A. Sen, and V. Garg. Solving computation slicing using predicate detection. *IEEE Trans. Parallel Distrib. Syst.*, 18(12):1700–1713, Dec 2007.

[24] A. Sen and V. Garg. Formal verification of simulation traces using computation slicing. *IEEE Trans. on Computers*, 56(4):511–527, Apr 2007.

[25] J. Strejcek. *Linear temporal logic: Expressiveness and model checking*. PhD thesis, Faculty of Informatics, Masaryk University in Brno, 2004.

[26] A. I. Tomlinson and V. K. Garg. Monitoring functions on global states of distributed programs. *J. of Parallel and Distrib. Comput.*, 41(2):173–189, 1997.

[27] C. Xu and S. C. Cheung. Inconsistency detection and resolution for context-aware middleware support. In *Proc. ACM SIGSOFT Intl. Symp. on Foundations of Softw. Eng. (FSE'05)*, pages 336–345, Sep 2005.

[28] C. Xu, S. C. Cheung, W. K. Chan, and C. Ye. Partial constraint checking for context consistency in pervasive computing. *ACM Trans. Softw. Eng. Methodol.*, 19(3):1–61, 2010.

[29] Y. Yang, Y. Huang, J. Cao, X. Ma, and J. Lu. *Design of a Sliding Window over Asynchronous Event Streams*. Technical Report ICS-NJU-092711, Institute of Computer Software, Nanjing University, Nov 2011. http://arxiv.org/abs/1111.3022.

[30] J. Yu, Y. Huang, J. Cao, and X. Tao. Middleware support for context-awareness in asynchronous pervasive computing environments. *IEEE/IFIP Intl. Conf. on Embedded and Ubiquitous Computing*, 0:136–143, 2010.