
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 dynamic properties, and detection of the specified temporal evolution of environment state, specification to sections, formulas, figures, tables, algorithms, and references are referring to entities in this supplementary file.

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

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 →) 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: \( e_0^{(k)}, s_0^{(k)}, e_1^{(k)}, s_1^{(k)}, e_2^{(k)}, s_2^{(k)}, \ldots \). 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 \( P^{(j)} \); and
- \( VC^{(j)}[j] \) 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 $\Phi$ can be specified as:

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

with $\emptyset$ denoting the empty set, $\epsilon$ denoting an empty word, and $\epsilon \in \Sigma$.

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

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

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

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

## 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 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 $s_{i^{(k)}}$, it sends a checking message to $P_{che}$.

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

1. Upon $LP^{(k)}$ becomes true
2. send control message($VC^{(k)}$) to each $P^{(j)}(j \neq k)$;
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 $n$
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 $\varphi = Pos(\Phi) \lor Def(\Phi)$;
3. extend $\Phi$ into $\Phi'$; transform $\Phi'$ into DFA $A(\Phi')$;
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($Act(LAT), A(\Phi')$); /* 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 $P^{(k)}$ are listed in Algorithm 1. During the initialization, we first get the property $Pos(\Phi)$ or $Def(\Phi)$, extend $\Phi$ to $\Phi'$, and obtain the DFA $A(\Phi')$. When a new local state $s_{i^{(k)}}$ arrives, we maintain the active surface and check the specified
Algorithm 3: grow_new(A(LAT), s_{i}^{(k)})
1 add s_{i}^{(k)} to Que^{(k)}; update g_{max};
2 combine C_{max} and s_{i}^{(k)} to get a global state G;
3 if G is C_{max} then
4 connect C_{max} to G; grow(G);
5 Act(LAT) = \{C \mid C \in LAT, \exists k, C[k] = g_{max}[k]\};

subroutine grow(C)
1 prec(C) = \{C' \mid \forall i, C'[i] \in Que^{(i)}, C' \text{ is CGS}, C' \prec C\};
2 sub(C) = \{C' \mid \forall i, C'[i] \in Que^{(i)}, C' \text{ is CGS}, C \prec C'\};
3 if sub(C) = \emptyset then C_{max} = C;
4 foreach C' in prec(C) do
5 if C' \notin LAT then
6 connect C' to C; grow(C');
7 foreach C' in sub(C) do
8 if C' \notin LAT then
9 connect C to C'; grow(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 C_{max} to obtain a global state G, as shown in line 2 of Algorithm 3. The lattice can grow, iff 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(C). Theorem 3 in the main file ensures that the active surface is sufficient for the growth of new CGSs. During the growing process, 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(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 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(A(LAT), A(\Phi'))
1 compute_reachable_states(C_{max});
2 Boolean flag = true;
3 Set_accept = A(\Phi').getAcceptingStates();
4 foreach C in Act(LAT) do
5 Set_reach = C.getReachableStates();
6 if Set_reach \cap Set_accept \neq \emptyset then
7 Pos(\Phi') is true; /* Pos(\Phi') return */
8 if Set_reach \subseteq Set_accept then
9 flag = false; /* Def(\Phi') break */
10 if flag = true then Def(\Phi') is true;

subroutine compute_reachable_states(C)
1 label_CGS_predicates(C);
2 prec(C) = \{C' \mid C' \in LAT, C' \prec C\};
3 foreach CGS C' in prec(C) do
4 if C'.getReachableStates() = \emptyset then
5 compute_reachable_states(C');
6 R_{\Phi'}^{C'} = \bigcup_{C' \in prec(C)} R_{\Phi'}^{C'}(C');
7 foreach state q in R_{\Phi'}^{C} do
8 foreach letter a in C.getLabeling() do
9 if \delta(q, a) \in C.getReachableStates() then
10 C.addReachableState(\delta(q, a));

Algorithm 5: prune_old(A(LAT))
1 foreach CGS C in LAT do
2 Boolean flag = false;
3 for i = 1 to n do
4 if C[i] = g_{max}[i] then
5 flag = true; /* C is active CGS */ break;
6 if flag = false then /* inactive CGSs are deleted */
7 delete C;

4.3 Complexity Analysis

First, we discuss the construction of the DFA. Given a regular expression \Phi, we can get the corresponding NFA in O(m), where m is the length of \Phi. 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 \Phi' (\Phi' = \Sigma^{*}\Phi\Sigma^{*}) is never greater than that of the NFA corresponding to \Phi (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.

![System architecture of MIPA](image)

Fig. 1. System architecture of MIPA

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. 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 $P_{\text{prob}}$, the Response Time of property detection $Cost_t$, and the Space Cost of property detection $Cost_s$. $P_{\text{prob}}$ is calculated as the ratio of $N_{\text{SurfMaint}}$ to $N_{\text{physical}}$. Here, $N_{\text{SurfMaint}}$ denotes the number of times SurfMaint detects the specified dynamic property. $N_{\text{physical}}$ denotes the number of times such property holds in the physical world. $Cost_t$ denotes the time from the instant when $P_{\text{che}}$ 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 $P_{\text{prob}}$, 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), $P_{\text{prob}}$ 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.
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.

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 $\text{Pos}$ or $\text{Def}$ in front. Thus, the expressive power of dynamic properties and CTL are overlapping but not contained by each other.

### References


