Enabling Context-awareness by Predicate Detection in Asynchronous Pervasive Computing Environments

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

Abstract—Pervasive applications are involving more and more autonomous computing and communicating devices, augmented with the abilities of sensing and controlling the logical / physical environment. To enable context-awareness for such applications, we are challenged by the intrinsic asynchrony among the context collecting devices. To this end, we introduce the predicate detection theory and propose the Predicate-Detection-based Context-Awareness (PD-CA) framework, in which: a) logical time is used to explicitly cope with the asynchrony; b) specification of predicates enables the applications to express contextual properties of their concerns; c) online and incremental predicate detection algorithms effectively enable context-awareness at runtime. Under the guidance of the PD-CA framework, we present the design and implementation of the MIPA middleware, which shields the applications from the burden of processing the asynchronous contexts. We also demonstrate how PD-CA simplifies the development of context-aware applications. Experimental evaluations show the performance of MIPA in supporting context-aware applications despite of the asynchrony.

Index Terms—Context-awareness, predicate detection, asynchrony.

1 INTRODUCTION

Pervasive applications are undergoing changes as more and more mobile devices are augmented with sensing and controlling abilities, besides the basic abilities of computation and communication. We call such devices C³S (Computation, Communication, Control, and Sensing) devices. Examples of C³S devices include mobile robots patrolling in a chemical plant for safety management [1], [2] and smart phones equipped with a variety of sensors [3], [4].

C³S devices can provide rich context information for the applications, and pervasive applications are typically designed to be context-aware, i.e., intelligently adapting their behavior to the environment [5], [6], [7]. However, enabling context-awareness through C³S devices is faced with severe challenges, as detailed below.

The contexts of interest to a pervasive application often span a geographically large area, and contain rich semantics. This is often beyond the ability of one single C³S device. Thus, a group of autonomous but also coordinating C³S devices should be deployed. Take a chemical plant scenario for example. A group of mobile robots are deployed to periodically patrol the plant for safety management [1], [2]. The robots need to proceed in certain formation to cover all possible spots of hazardous material leak. Appropriate spreading of multiple robots can also enable the robots to collect contexts with better quality, e.g., to sense the average temperature in the plant.

The coordination among the C³S devices is intrinsically asynchronous. There is no global clock available among the C³S devices. Constrained resources and task scheduling of the C³S devices (often embedded systems) may lead to unpredictable computation delay [8]. The growing adoption of wireless communications, which are prone to bandwidth shortage, network congestion, unpredictable routings, and retransmission, leads to unpredictable communication delay [9], [10], [11], [12]. All these characteristics of the C³S devices and their communication networks lead to the intrinsic asynchrony among the contexts they collect [12], [8].

One possible solution to cope with the asynchrony is clock synchronization. However, clock synchronization may not enable correct and fault-tolerant coordination among the autonomous C³S devices [11], [13], [2]. Thus, it cannot enable context-awareness despite of the asynchrony in pervasive scenarios enriched of coordinating C³S devices. Specifically, each C³S device only has its own local clock, which cannot be perfectly synchronized [10], [14], [15]. The uncertainty caused by the skew among the clocks may lead to incorrect behavior [8]. Besides, clock synchronization schemes make assumptions on process execution speeds and communication delay [10]. These assumptions may not be guaranteed for the autonomous C³S devices. A group of robots are prone...
to incorrect behavior even if one single assumption is violated \cite{11}. The inaccuracy of synchronization and the potential violation of assumptions make reasoning based on time and timeouts a delicate and error-prone undertaking \cite{11}. Furthermore, periodic clock synchronization may be unaffordable in terms of energy consumption, or be hampered due to device autonomy and administrative boundaries such as privacy concerns and security issues \cite{8}, \cite{12}. Consequently, it is more practical to have few or, better, no synchrony assumption in scenarios of coordinating C$^3$S devices \cite{2}, \cite{8}, \cite{11}.

To enable context-awareness for pervasive applications enriched with C$^3$S devices, we introduce the predicate detection theory and propose the Predicate-Detection-based Context-Awareness (PD-CA) framework, which consists of three essential parts:

- Logical time is used to cope with the asynchrony among contexts collected from the system of autonomous C$^3$S devices. Temporal orders among asynchronous contextual events are encoded and decoded via the logical vector clock. Global snapshot of the asynchronous system of C$^3$S devices is redefined under the notion of logical time. Dynamic behavior of the C$^3$S devices are modeled over sequences of global snapshots.

- Specification of predicates enables the applications to express their concerns on properties of the contexts. Based on the modeling above, the specification can delineate local contextual properties on local states of one C$^3$S device, global contextual properties on snapshots of the system of C$^3$S devices, and dynamic behavioral properties on sequences of snapshots.

- Context-awareness is enabled by detection of the specified contextual property at runtime. The detection is such a persistent process that new emerging contexts trigger the incremental detection of the specified property.

Under the guidance of the PD-CA framework, we develop the Middleware Infrastructure for Predicate detection in Asynchronous environments (MIPA) \cite{16}. In context-aware computing scenarios, MIPA first receives contextual properties from the applications. It then decomposes the global contextual properties to local ones, with which MIPA instructs each C$^3$S device to collect the related contexts. MIPA detects the specified contextual properties with the contexts in an online and incremental manner and informs the applications when the properties are satisfied. MIPA adopts a layered architecture to support this context processing process.

We further discuss how MIPA shields the applications from the burden of coping with asynchronous contexts. Based on the PD-CA framework, the context-aware adaptation logic of the application is constructed in a condition-action manner. The contextual property serves as the condition of context-aware behavior. Experimental evaluations show the performance of MIPA in supporting context-aware applications despite of the asynchrony.

The rest of this work is organized as follows. Section 2.2 presents a motivating example. Section 3 overviews preliminaries of the predicate detection theory. Section 4 presents our PD-CA framework. Section 5 discusses the design of MIPA. Section 6 discusses how MIPA simplifies the development of context-aware applications. Section 7 presents the experimental evaluation. Section 8 discusses the related work. In Section 9, we conclude the work and discuss the future work.

2 Motivations and Challenges by Example

To further justify our PD-CA framework, we first introduce a case study scenario. Then we discuss the motivations and challenges in detail based on this scenario.

2.1 Chemical Plant Safety Management Scenario

Let us consider a safety management application of a chemical plant, as shown in Fig. 1. Several mobile robots are deployed to periodically patrol the plant. Robots are equipped with temperature sensors and hazardous material leak sensors for safety management. To ensure the safety of the plant, the application is concerned with the following contextual properties:

- $\phi_1$: all the robots detect hazardous material leak in a workshop.
- $\phi_2$: the average temperature in a workshop gets exceptionally high.

When $\phi_1$ is detected, the safety management application should sound the alarm, notify the workers in the workshop to escape, and notify the plant safety officer nearby to cope with the leak. When $\phi_2$ is detected, the safety management application should notify the fireman nearby to cope with the possible fire accident.

To ensure the quality of surveillance, the application is also concerned with the status of the coordination among the robots. For example, the application is concerned with whether the robots proceed in the right formation (to evenly cover a workshop). It is also concerned with whether the robots pass the gateway one by one in certain order (since only one robot can pass the gateway at the same time, the robots need to coordinate to pass the gateway one by one). Specifically, the application is concerned with the following properties:

- $\phi_3$: robot R1 or R3 cannot detect R2 and R2 cannot detect R1 or R3.
- $\phi_4$: the robots pass the gateway one by one, in the order of R1, R2, and R3.
to fulfil their tasks. In order to achieve context-awareness through a group of asynchronously coordinating mobile robots, there are mainly three critical challenges.

**Challenge 1. How to interpret the asynchronous contexts?** The context model needs to explicitly cope with the asynchrony among contexts. Specifically, the application is often concerned with global properties of the environment, while the related contexts are generated from the distributed mobile robots. Because the robots coordinate in an asynchronous manner, the contexts generated from them are also asynchronous.

Take property $\phi_3$ in Section 2.1 as an example. Without global clocks, we cannot tell whether the three pieces of contexts “R1 cannot detect R2”, “R3 cannot detect R2”, and “R2 cannot detect R1 or R2” occur concurrently. Existing context modeling techniques mainly assume the availability of global clocks, but cannot cope with distributed and asynchronous contexts [18], [19], [20]. Thus, the precise modeling of distributed and asynchronous contexts is required, which enables further specification and detection of contextual properties the application is concerned with.

**Challenge 2. How to enable the application to express its concerns on the asynchronous computing environment?** A specification formalism is required, which can enable the application to delineate its concerns on the asynchronous contexts, and inform the related robots of their tasks of context acquisition. Specifically, the application may specify various types of contextual properties. The properties may describe status of the physical environment, such as hazardous material leak and indoor temperature. The properties may also describe status of the asynchronous coordination among the robots, such as whether the robots correctly pass the gateway. Based on the specification, we can accurately and conveniently tell the robots what contexts they should collect. The application is then shielded from the underlying details of acquisition and processing of asynchronous contexts.

**Challenge 3. How to efficiently detect changes of the computing environment?** Online and incremental detection algorithms are required to detect the contextual properties. As shown in the scenario, the application has close interaction with the robots at runtime. The detection should also be finished as soon as possible, when the property of interest to the application occurs. Thus, the detection algorithms should be incremental, only checking the newly collected contexts. Processing all the contexts each time is cost-ineffective and often unnecessary, and should be avoided.

### 3 Preliminaries of Predicate Detection

The PD-CA framework relies on the theory of predicate detection [21], [22], [23], [24], [25], [26], [27], [28], [29].
Predicate detection refers to the checking of global predicates over asynchronous computations [23]. Predicate detection consists of three essential parts: 1) modeling of the asynchronous computation; 2) specification of global predicates; 3) detection of specified predicates [28, 12]. We outline these three parts below.

3.1 Modeling of the Asynchronous Computation

A distributed system consists of a collection of instrumented processes $P^{(1)}, P^{(2)}, \ldots, P^{(n)}$. Examples of instrumented processes include a thread in a program or a software process manipulating a C/S device. One checker process $P_{ch}$ is in charge of collecting the trace of the distributed system execution and detecting the specified predicate.

3.1.1 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. Different processes may run at different speeds. We assume that dedicated message protocols are employed to ensure that no messages are lost, altered, or spuriously introduced [23, 26].

As the system executes, each $P^{(k)}$ generates its (potentially infinite) trace of local states connected by events: $s^{(k)}_{0}, f^{(k)}_{1}, s^{(k)}_{1}, f^{(k)}_{2}, \ldots$. The order among the events of instrumented processes is encoded by the logical vector clock [21] which indicates the happen-before relation (denoted by ‘$\prec$’) resulting from message causality [17]. Detailed definitions of the happen-before relation and the logical vector clock can be found in [23, 24, 28, 12, 26, 27]. For example, we have $s^{(1)}_{0} \rightarrow s^{(2)}_{2}$ and $s^{(2)}_{0} \rightarrow s^{(1)}_{1}$ in Fig. 2.

3.1.2 Space-Time Diagram and Lattice of Snapshots

Global snapshots of the system can be depicted by the consistent cuts [23] over the STD. Snapshot sequences can be depicted by the consequent cuts over the STD.

One key notion in predicate detection is that the set of observed snapshots over the trace with the ‘$\prec$’ relation has the lattice structure (LAT) [24, 23]. For example, Fig. 2 shows the LAT of snapshots corresponding to the STD. The dots denote snapshots, crosses (‘$\times$’) denote inconsistent global states, and edges depict the ‘$\prec$’ relation. The bold line is a possible snapshot sequence.

In the LAT, we can see the detailed evolution of each process, but it is not convenient to reason over the global snapshots or snapshot sequences of the whole system. In comparison, global snapshots are dots in the LAT, and snapshot sequences are paths of consequent dots in the LAT. Though the LAT is more efficient to illustrate the global behavior of the system, it is usually computationally expensive to construct the LAT [22, 23]. The STD and the LAT are different illustrations of the same computation, and should be used according to different needs.

Due to the uncertainty caused by the asynchrony, we can obtain multiple snapshot sequences from the LAT delineating all possible executions of the system. We only know that the actual execution of the system is one of these sequences but never know which one [23, 26, 27, 28, 12, 31].

3.2 Specification of Global Predicates

Formal specification of predicates expresses the users’ concerns on properties of the system. Formal specifications are precise and have the advantage of being amenable to automatic analysis and manipulation [32]. Based on the modeling of the system behavior discussed above, we can specify two classes of predicates: snapshot predicates and sequence predicates [28]. Snapshot predicates are properties that can be evaluated over a snapshot of the system, while sequence predicates are properties that can be evaluated over dynamic system behavior, i.e., the snapshot sequence, as shown in Fig. 2. For example, properties $\phi_1$, $\phi_2$, and $\phi_3$ in Section 2.1 can be specified as snapshot...
predicates, and \( \phi_4 \) can be specified as a sequence predicate. In what follows, we briefly introduce these two classes of predicates.

### 3.2.1 Snapshot Predicates

Snapshot predicates can be evaluated based on the information with respect to one snapshot. We mainly focus on two types of snapshot predicates below.

**Conjunctive predicates** delineate the concurrency among local activities (which are delineated by local predicates) on each process. They are widely used, and can also serve as the basis for the specification of many other types of predicates [12]. Conjunctive predicates are defined as follows:

\[
CP := \Phi \mid \text{Def}(\phi)
\]

\[
\phi := \phi^{(1)} \land \phi^{(2)} \ldots \land \phi^{(n)}
\]

\[
\phi^{(k)} := \text{local predicate on } p^{(k)} (1 \leq k \leq n)
\]

where the value of \( \phi^{(k)} \) only depends on the local variable \( x^{(k)} \) at \( p^{(k)} \), and modal operators \( \text{Pos()} \) and \( \text{Def()} \) are adopted to cope with the multiple possible snapshot sequences resulting from the asynchrony discussed in Section 3.1.2 [22]. Informally, \( \text{Pos}(\phi) \) means that predicate \( \phi \) holds on one possible snapshot sequence, while \( \text{Def}(\phi) \) means that predicate \( \phi \) holds on all possible snapshot sequences. Detailed discussions on the semantics can be found in [12].

Property \( \phi_1 \) in Section 2.1 can be expressed as a conjunctive predicate “\( \text{Pos}(R_1) \text{ detects the leak} \land R_2 \text{ detects the leak} \lor R_3 \text{ detects the leak})\”.

The other important type of snapshot predicates is the relational predicate, which is defined as follows:

\[
RP := \Phi \mid \text{Def}(\phi)
\]

\[
\phi := x^{(1)} + x^{(2)} + \ldots + x^{(n)} \text{ relop } C
\]

\[
x^{(k)} := \text{local variable at process } p^{(k)} (1 \leq k \leq n)
\]

\[
\text{relop} := < | > | \leq | \geq |
\]

where \( C \) is a constant. Detailed discussions on the semantics can be found in [33].

Property \( \phi_2 \) in Section 2.1 can be expressed as a relational predicate “\( \text{Pos}(\text{temperature}_R1 + \text{temperature}_R2 + \text{temperature}_R3 > 3 \times \text{threshold})\)”.

### 3.2.2 Sequence Predicates

Sequence predicates can be evaluated based on the information on a snapshot sequence. In this section, we present two important types of sequence predicates, regular expression predicates and temporal logic predicates, which can be employed to characterize behavioral patterns of the asynchronous computation.

Regular expression predicates can be specified as follows [28, 27]:

\[
REP := \Phi \mid \text{Def}(\Phi)
\]

\[
\Phi := \emptyset \mid \varepsilon \mid a \mid \Phi \lor \Phi \lor \Phi \lor \Phi
\]

where \( \emptyset \) denotes the empty set, \( \varepsilon \) denotes an empty word, and \( a \in \Sigma \). The alphabet \( \Sigma \) is the set of snapshot predicates involved in the regular expression predicates. Regular expression predicates unify a large body of important predicates including linked predicates [34], interval-constrained sequence predicates [35], etc. Detection of regular expression predicates can boil down to a language recognition problem [28, 27]. Detailed discussions on the semantics can be found in [28].

Property \( \phi_4 \) in Section 2.1 can be expressed as a regular expression predicate “\( \text{Def}(a \ast ab + bc \ast cd + d)\)” with \( a \) indicates that \( R_1, R_2, \) and \( R_3 \) are all in workshop \( A \), \( b \) indicates that \( R_2 \) and \( R_3 \) are in workshop \( A \) while \( R_1 \) is in workshop \( B \), \( c \) indicates that \( R_3 \) is in workshop \( A \) while \( R_1 \) and \( R_2 \) are in workshop \( B \), and \( d \) indicates that \( R_1, R_2, \) and \( R_3 \) are all in workshop \( B \).

An important type of temporal logic predicates can be specified as CTL formulae, which can effectively express the notion of branching time. CTL has a two-stage syntax where formulae in CTL are classified into state and path formulae [31]. The state formulae are assertions about the atomic propositions (i.e., snapshot predicates) in the states (i.e., snapshots) and their branching structure, while path formulae express temporal properties of paths (i.e., snapshot sequences).

CTL state formulae over the set \( \Sigma \) of atomic propositions are formed according to the following grammar:

\[
\Phi := T \mid a \mid \Phi_1 \land \Phi_2 \mid \neg \Phi \mid \exists \varphi \lor \forall \varphi
\]

where \( a \in \Sigma \) and \( \varphi \) is a path formula. CTL path formulae are formed according to the following grammar:

\[
\varphi := \bigcirc \Phi \mid \Phi_1 \lor \Phi_2
\]

where \( \Phi, \Phi_1, \) and \( \Phi_2 \) are state formulae. Detailed discussions on the semantics can be found in [31].

### 3.3 Detection of Specified Predicates

The predicate detection algorithms often need to be online, which is decided by the application scenarios. For example in a context-aware computing scenario, the application has close interaction with the computing environment and needs to get informed as soon as possible when the predicate of its concern turns true. We design online predicate detection algorithms for such scenarios.

The detection algorithms also need to be incremental, i.e., only the newly updated part of the trace of the distributed system is checked. The trace of the distributed system continuously evolves as the system runs. Each time the trace evolves (i.e., a new local state arrives), the checker process \( P_{che} \) should check whether the new part of the trace changes the value of the specified predicate.

In principle, we can design a general algorithm to detect all types of predicates defined in Section...
In this section, we first describe the essential concepts in the Predicate-Detection-based Context-Awareness (PD-CA) framework. Then we discuss how these concepts facilitate the development of context-aware applications.

A context-aware computing scenario adopts a middleware-based architecture. Context collecting devices send local contexts to the context-aware middleware. The middleware processes the contexts, evaluates the contextual properties specified by the application, and notifies the application to conduct context-aware behavior accordingly. This architecture will be detailed in Section 5 and 6.

The PD-CA framework solves the challenges in Section 2.3 and enables context-awareness based on the predicate detection theory. The mapping of concepts between the predicate detection theory and the PD-CA framework is listed in Table 1.

### Table 1

| Concepts in PD | Concepts in PD-CA |
|---------------|-------------------|
| instrumented process | context collecting process on the context collecting device |
| local variable | type of local context provided by the context collecting device |
| local event | contextual event: value change of local variable/predicate |
| local state | local context (value of local variable/predicate) with logical timestamp |
| global snapshot | global snapshot of the system of context collecting devices |
| snapshot sequence | dynamic behavior of the system of context collecting devices |
| local predicate | local contextual property of a context collecting device |
| snapshot predicate | global instantaneous property of the system of context collecting devices |
| sequence predicate | dynamic behavioral property of the system of context collecting devices |
| checker process | property detection process on the property detection server |
| detection of predicates | persistent monitoring of contextual properties |

### 4 The PD-CA Conceptual Framework

In this section, we first describe the essential concepts in the Predicate-Detection-based Context-Awareness (PD-CA) framework. Then we discuss how these concepts facilitate the development of context-aware applications.

As discussed in Section 2.1, a context-aware application is often concerned with and reacts to the global properties of its contexts. While one single context collecting device can only provide limited types of contexts, the contexts of an application are often provided by a group of coordinating context collecting devices. These context collecting devices persistently collect local contexts and send them to the context-aware middleware. The middleware then combines these local contexts into global contexts for the detection of global properties specified by the application.

In our PD-CA framework, the coordinating context collecting devices are modeled as context collecting processes, and form an asynchronous system of context collecting processes. The devices coordinate by exchanging messages. The type of local context provided by a device is modeled as a local variable. The value of the local context and its logical timestamp define a local state of the device. Value changes of local contexts are modeled as contextual events.

Based on our intuitive understanding of the computing environment, we model the spatially distributed context as a hierarchy of global-local context. Local contexts are defined on local states of context collecting devices, while global contexts are defined on global system snapshots (i.e., vectors of local states of each context collecting device). In our PD-CA framework, the hierarchy of specification is based on the hierarchy of global-local context, as detailed in Section 4.2.

### 4.1 Modeling of Asynchronous Context

The key problem in the modeling of asynchronous context is to cope with the asynchrony originating from the mismatching between the capability-constrained context collecting devices and the rich semantics of the computing environment. The contexts are spatially distributed and temporally asynchronous. We discuss these two characteristics in detail.

#### 4.1.1 Modeling the Spatially Distributed Context

As discussed in Section 2.1, a context-aware application is often concerned with and reacts to the global properties of its contexts. While one single context collecting device can only provide limited types of contexts, the contexts of an application are often provided by a group of coordinating context collecting devices. These context collecting devices persistently collect local contexts and send them to the context-aware middleware. The middleware then combines these local contexts into global contexts for the detection of global properties specified by the application.

In our PD-CA framework, the coordinating context collecting devices are modeled as context collecting processes, and form an asynchronous system of context collecting processes. The devices coordinate by exchanging messages. The type of local context provided by a device is modeled as a local variable. The value of the local context and its logical timestamp define a local state of the device. Value changes of local contexts are modeled as contextual events.

Based on our intuitive understanding of the computing environment, we model the spatially distributed context as a hierarchy of global-local context. Local contexts are defined on local states of context collecting devices, while global contexts are defined on global system snapshots (i.e., vectors of local states of each context collecting device). In our PD-CA framework, the hierarchy of specification is based on the hierarchy of global-local context, as detailed in Section 4.2.

#### 4.1.2 Modeling the Temporally Asynchronous Context

As discussed in Section 2, the distributed context collecting devices coordinate in a loosely-coupled asynchronous manner. Thus the local contexts collected by them are intrinsically asynchronous. Without global clocks, we cannot easily tell whether a group of local contexts collected from different devices forms a meaningful global context.

In our PD-CA framework, due to the asynchrony, we do not rely on global clocks, but on logical time to model the temporal order among the local states of different context collecting devices. The temporal order is defined by the happen-before relation resulting from message exchanges between the context collecting devices (defined in Section 3.1.1). Local states from different context collecting devices, which have no happen-before relation among each other, form a meaningful snapshot, as introduced in Section 3.1.2. The local contexts of a meaningful snapshot form a meaningful global context. As the context collecting devices persistently collect contexts, we can get sequences of meaningful snapshots. Since the happen-before relation is a partial order relation, we can observe multiple possible snapshot sequences, but
cannot tell which is the actual one. This uncertainty caused by the asynchrony is explicitly modeled in the STD or the LAT (in Section 3.1.2).

### 4.2 Specification of Contextual Properties

In our PD-CA framework, the context-aware application expresses its concerns on contextual properties of the computing environment by specification of (usually global) predicates. The hierarchical specification of predicates is directly based on the hierarchy of global-local context, as discussed in Section 4.1.1.

#### 4.2.1 The Hierarchy of Predicates

We introduce the hierarchy of predicate specification in a bottom-up manner, and illustrate it in Fig. 2.

- **Local predicates.** Local predicates are specified over local states of context collecting devices, to indicate the application’s concern on specific aspects of the computing environment. For example, “R1 detects the leak of hazardous material” (in property $\phi_1$ in Section 2.1) is a local predicate.

- **Snapshot predicates.** Snapshot predicates are specified over global snapshots (as discussed in Section 3.2.1), to indicate the application’s concern on global instantaneous properties of the computing environment. For example, “the average temperature is greater than the threshold” (property $\phi_2$ in Section 2.1) is a snapshot predicate.

- **Sequence predicates.** Based on the modeling of temporal evolution of context, the application can specify sequence predicates to describe the dynamic properties and behavioral patterns of the computing environment on snapshot sequences. For example, “the robots pass the gateway in certain order” (property $\phi_3$ in Section 2.1) is a sequence predicate.

- **Modal operators.** To cope with the uncertainty resulting from the asynchrony, modal operators $\text{Pos}(\cdot)$ and $\text{Def}(\cdot)$ are adopted to interpret the predicates over the multiple possible snapshot sequences, as discussed in Section 3.2.1. Each of the two modal operators has its own characteristic. $\text{Pos}(\cdot)$ is often used to eliminate the possibility of “bad things”, while $\text{Def}(\cdot)$ is often used to ensure the occurrence of “good things”. For example, we use the modal operator $\text{Pos}(\cdot)$ in property $\phi_3$ (formalized in Section 3.2.1), since we need to prevent fire accidents in all possible snapshot sequences.

#### 4.2.2 Separation of Concerns

The specification of contextual properties greatly facilitates the separation of concerns among the application, the context collecting devices, and the middleware, as shown in Fig. 3.

Contextual properties specified in predicates provide a logical abstraction of the computing environment (i.e., the asynchronous system of context collecting devices) and hide the complexity of low-level programming to the application. In this way, the application only have to specify predicates to the middleware and react to the satisfaction of the predicates, not having to know how the asynchronous contexts are collected and processed. The middleware collects the related asynchronous contexts, detects the specified predicates, and notifies the application when the predicates turn true.

As the specified global property is often in the form of a hierarchy of snapshot-local predicates, the middleware can subscribe consisting local predicates to the corresponding context collecting devices. Rather than sending all the collected low-level contexts (values of each local variable $x(k)$ in our PD-CA framework) to the middleware, each context collecting device detects the subscribed local predicate, and sends the filtered local contexts (values of the local predicate) to the middleware when the value of the predicate changes. The middleware processes the aggregated local contexts and detects the specified predicate under the guidance of the PD-CA framework. It does not have to know how the local contexts are collected and how the local predicates are evaluated.

### 4.3 Detection of Specified Properties

In the detection of specified properties, the checker process is instantiated as a property detection process deployed on the middleware, which detects the specified predicates in an online and incremental manner.

Since context-awareness is such a persistent process that new emerging contexts trigger the application to conduct context-aware behavior, our detection of specified predicates is achieved in an event-driven manner. When new contexts are generated by a context collecting device, they will first be filtered by a local predicate, modeled as contextual events (indicating the value changes of the local predicate), and sent to the corresponding property detection process on the middleware. When receiving a contextual event, the property detection process detects whether the newly arrived context makes the specified predicate true and notifies the application accordingly.
Though specific predicate detection algorithms are designed for different types of predicates (as discussed in Section 3.3), predicate detection in our PD-CA framework is achieved in two steps generally. The property detection process first maintains the STD or the LAT of global contexts, and then invokes the corresponding predicate detection algorithm to detect the specified predicate over the STD or the LAT. The maintenance of the STD or the LAT is generally reusable and independent to the detection of specific types of predicates. Implementation details of the predicate detection algorithms will be discussed in Section 5.4.

4.4 Design Process of PD-CA

After discussing essentials of the PD-CA framework, we further discuss the process of developing context-aware applications using the framework. Based on predicate detection, context-awareness in asynchronous environments is achieved by runtime detection of contextual properties. The middleware receives the contextual property from the application, persists collected contexts from multiple distributed context collecting devices, detects whether the property holds, and notifies the application accordingly. Specifically, the process of developing a context-aware application using the PD-CA framework includes five steps, as detailed below.

Step 1. Obtaining the specification. The application specifies the contextual property to the middleware. The middleware then parses the property, identifies the consisting local predicates in the property, finds the related context collecting devices, registers the local predicates to their corresponding context collecting devices, and launches the property detection process which is in charge of detecting the contextual property.

Step 2. Acquisition of the asynchronous context. Each context collecting device registers itself to the middleware with the type of context it provides. When receiving the registration of a local predicate, the context collecting process on the context collecting device filters local collected contexts from the local predicate and generates local states. When the context collecting devices exchange messages between each other, the logical vector clocks are updated by piggybacking logical timestamps on messages [12], [28]. When a context collecting device sends/receives messages or the value of the local predicate changes, the context collecting process sends the local state (local context with current logical timestamp) to the property detection process on the middleware.

Step 3. Processing of the asynchronous context. When the property detection process on the middleware receives the local contexts from distributed context collecting processes, it first explicitly maintains the STD or the LAT based on the causal relation among these asynchronous contexts, and then detects the specified contextual property.

Step 4. Notifying the context-aware application. If the contextual property is evaluated true, the middleware notifies the context-aware application of the satisfaction to conduct context-aware behavior.

Step 5. Taking context-aware behavior. The context-aware behavior of the application is constructed around the contextual property. When notified of the satisfaction of the property, the application conducts the corresponding context-aware behavior.

By providing middleware support for this design process, the development of context-aware applications can be greatly simplified. The application developers can be relieved of the time- and energy-consuming burden of collecting asynchronous contexts from distributed heterogeneous context collecting devices and processing these asynchronous contexts, as in Step 2-4. The application developers only need to specify contextual properties and implement the corresponding context-aware behavior.

In the following Section 5 and 6 we discuss the design and implementation of the middleware and the application, respectively.

5 MIPA - Providing Middleware Support for the PD-CA Framework

In this section, we present the design and implementation of MIPA – Middleware Infrastructure for Predicate detection in Asynchronous environments, which would greatly simplify the tasks of building context-aware applications with the PD-CA framework [12], [28], [31], [36], [16].

The design process of the PD-CA framework (in Section 4.4) consists of several logically independent steps. Thus we adopt a layered architecture for MIPA. The layered architecture groups related functionalities into distinct layers and provides software engineering benefits such as separation of concerns, information hiding, extensibility, and reusability.

The architecture of MIPA is listed in Fig. 4. The middleware can be deployed as a centralized property detection server (components in dark grey color), and several context collecting clients (components in light grey color) over context collecting devices. The main functions of each layer are explained below:

- The specification management layer is in charge of obtaining the contextual property from the application, locating the corresponding context collecting devices, and maintaining the group of processes involved in the detection of the property.
- The context acquisition layer is in charge of collecting asynchronous contexts on context collecting devices and sending them to the property detection server for predicate detection.
The property detection layer receives the local contexts sent from the context collecting devices and detects the specified contextual property. When the contextual property is detected true, the property detection layer will notify the application.

We first overview the system components. Then, we discuss the design and implementation of each layer.

5.1 System Components

To support the components involved in the processing of asynchronous contexts and ensure a strong decoupling among them, several system components are provided.

5.1.1 Broker

The broker is in charge of interacting with the application and context collecting clients, and delivering their requests to the corresponding components, as shown in Fig. 5. Specifically, the application registers the contextual property to the middleware through the broker and the broker delivers the property to the specification management layer for further processing. When the application unregisters the property, the broker notifies the specification management layer to stop the processes involved in the detection of the property. When the property is detected true, the middleware notifies the application through the broker. Similarly, the context collecting devices contact the broker to register/unregister themselves on the middleware.

5.1.2 Resource Manager

The resource manager is in charge of the management of the available types of contexts and their corresponding context collecting devices. It supports the registration/unregistration, as well as the lookup of context collecting devices.

5.2 Specification Management Layer

The specification management layer is in charge of the management of the specification and the processes involved in the detection of the specification. When a contextual property is received from the application through the broker, it should first be parsed and the related context collecting devices should be identified. Then, the corresponding processes, including the context collecting processes on the context collecting clients and the property detection process on the property detection server, are launched for the detection of the property. We detail the management of the specification and the group of processes involved in the predicate detection below.

5.2.1 Management of the Specification

The specification serves as the contract between the context-aware application and the context-aware middleware [32]. The predicates may have different types, such as conjunctive predicates and regular expression predicates (in Section 5.2.2). We provide a uniform mechanism to represent different types of predicates. Since sequence predicates are generally composed of several snapshot predicates, we extract these snapshot predicates as an alphabet to facilitate the description of sequence predicates. As the snapshot predicates are often in the form of hierarchies of snapshot-local predicates, our specification mechanism also adopts a hierarchical structure, as shown in Fig. 6.

Specifically, the “Snapshot predicates” node contains all consisting snapshot predicates as the alphabet, and each “Snapshot predicate” node defines a snapshot predicate. The “Contextual predicate” node contains the specified predicate which is composed of snapshot...
Each "Snapshot predicate" node is labeled with its type, such as conjunctive or relational, as introduced in Section 3.2.1. The "Contextual predicate" node is also labeled with its type, such as regular-expression or CTL, as introduced in Section 3.2.2. This structure can unify a large amount of typical predicates as introduced in Section 3.2 and is easy to extend new types of snapshot or sequence predicates.

We adopt XML which is a flexible way to create "self-describing data" and naturally has the hierarchical structure to realize our specification mechanism. The DTD of the specification is listed in Fig. 7. It currently supports conjunctive predicates, regular expression predicates, and CTL predicates. New types of predicates can be added by extending the "snapshotPredicate" and "contextualPredicate" in Fig. 7 as an example, property \( \phi_1 \) in Section 2.1 which is formalized in Section 3.2.1 is shown in Fig. 8.

When receiving the specification in XML, a predicate parser is employed to parse the specification into the structure shown in Fig. 6. Notice that different types of snapshot and sequence predicates have different structures. Thus, we provide different parsers for different types of predicates. For example, when the predicate parser reaches a snapshot predicate labeled "conjunctive" as shown in Fig. 6, the type is first recognized and the corresponding parser for conjunctive predicates is invoked to parse the predicate. Our design provides good reusability and extensibility. New types of predicates can be parsed by adding new parsers accordingly.

According to the type of the contextual predicate, the corresponding processes which are dedicated to the detection of the predicate can be further launched.

5.2.2 Management of the Group of Processes Involved in the Predicate Detection

After parsing the predicate, the consisting local predicates are identified. The middleware locates the context collecting devices corresponding to the local predicates through the resource manager, registers the local predicates to these devices (context collecting clients) to create context collecting processes which detect the local predicates, and creates a property detection process which detects the specified predicate on the property detection server.

These processes are distributed on different context collecting clients and the property detection server. To better maintain them, we organize the processes created for a predicate as a predicate detection group, and adopt a group manager to maintain the groups.

Specifically, when a predicate is registered/unregistered, the group of corresponding processes is created/destroyed. Notice that the context collecting processes may send messages between each other or to the property detection process. Thus, the property detection process should be launched before the context collecting processes, and the context collecting processes should not be started until they are all created. Likewise, during destruction, the context collecting processes should be destroyed before the property detection process. The creation and destruction are coordinated by the group manager.

5.3 Context Acquisition Layer

The context acquisition layer manages the registration/unregistration of context collecting devices, and acquires local contexts from them.
5.3.1 Management of Context Collecting Devices

With the development of embedded computing technologies, various new C³S devices are developed to support context-awareness. These devices register themselves to the middleware, coordinate to collect local contexts, and send local contexts to the middleware. Local contexts include information about the physical environment such as hazardous material leak and indoor temperature, as well as the logical environment delineating the status of coordination among the context collecting devices, such as the neighbor information.

To shield the application from low-level details of heterogeneous devices, all providers of all types of contexts are abstracted as context providers by implementing the same interface ContextProvider. Each device registers the types of local contexts it provides to the middleware by registering the corresponding context providers to the middleware. Therefore, the middleware can treat different types of local contexts from different devices in a similar way, which eases further processing of local contexts. To each context provider, a context collecting agent is assigned. These context collecting agents are in charge of manipulating the context providers and sending collected data from context providers to the context collecting processes.

To ease the integration of new context collecting devices, context collecting agents are implemented as plug-ins. When registering to the middleware, new devices register each context provider to the middleware using a context provider configuration in XML, i.e., a document containing the local context it provides and all the information needed to interact with the hardware or software modules of the device [37], [38]. The DTD of the XML for context provider configuration is listed in Fig. 9 and an exemplar configuration of a temperature sensor is listed in Fig. 10. As shown in Fig. 10, the temperature sensor provides the type of local context “temperature_R1”.

An ECA manager (introduced in Section 5.3.2 in detail) on each context collecting client is employed to manage the registration/unregistration of the context collecting agents. When a new device registers to the middleware, the ECA manager on the device parses all the local context provider configurations, and assembles the corresponding context providers (e.g., the class mipa.eca.sensor.Temperature in Fig. 10 for the temperature sensor of robot R1) and context collecting agents to interact with the context providers. The ECA manager then registers the types of local contexts to the resource manager on the property detection server (through the broker).

5.3.2 Acquisition of Local Contexts

To react to dynamic changes of the contexts, MIPA provides an efficient way to monitor the changes of contexts and to disseminate the changes to interested entities. As discussed in Section 4.2.2 the snapshot predicates often adopt the form of hierarchies of snapshot-local predicates, and the context collecting processes are only interested in the value changes of local predicates. We adopt the Event-Condition-Action (ECA) mechanism [39], [40], [18] to achieve persistent detection of local predicates, as shown in Fig. 11.

Specifically, when creating the group of processes dedicated to the detection of the specified predicate, the group manager locates the context collecting clients which provide the types of contexts required by the consisting local predicates, and registers each local predicate to the ECA manager of the corresponding context collecting client, as shown in Step 1 in Fig. 11. Then, the ECA manager creates an event listener (i.e., the context collecting process) which listens to the value changes of the local predicate, and registers the local predicate to the event filter, as in Step 2-3. In Step 4, the event filter takes the local predicate as the event condition on the corresponding context collecting agents. When the event filter receives the contexts pushed from the context collecting agents, it updates the value of the local predicate. When the value changes, the event filter notifies the context collecting process to take actions, as in Step 5-8. We support local predicates in first-order logic composed of atomic predicates defined on local contexts.

The context collecting process is in charge of taking actions to event changes, including updating local logical vector clock and sending local contexts with

Fig. 9. The DTD of the context provider configuration.

```xml
<!DOCTYPE contextProvider SYSTEM "contextProvider.dtd">
<contextProvider>
  <name>temperature_R1</name>
  <id>temperature_R1</id>
  <class>mipa.eca.sensor.Temperature</class>
  <location>nju/location>
  <valueType>Double/valueType>
  <dataSchema type="push"/>
  <dataDisseminate time="4" lazy="true"/>
  <arguments>
    <argument data="temperature_R1"/>
  </arguments>
</contextProvider>
```

Fig. 10. The context provider configuration for a temperature sensor of robot R1.
logical timestamps to the property detection process on the property detection server. Notice that the detection algorithms for different types of predicates may have different logics of context collecting processes [12], [28], [31], [26], but they share the same skeleton. Thus, we adopt the template method pattern [41] to facilitate the development of new types of context collecting processes. New context collecting processes integrate to the middleware by extending the abstract class AbstractNormalProcess, as shown in Fig. 12.

In method action() (i.e., Step 8 in Fig. 11), the concrete context collecting process receives the value changes of the local predicate, and it should send the local context with the current logical timestamp (currentClock in Fig. 12) to the property detection process. The methods onSendMsg() and onReceiveMsg() will be invoked when the local coordination algorithm on the device sends/receives messages to/from other context collecting devices. Thus, the concrete context collecting process should update the logical vector clock in these two methods, to capture the happen-before relation between the local contexts on different context collecting devices, as required in Step 2 of the design process in Section 4.4.

5.4 Property Detection Layer

The property detection layer detects the specified predicates and notifies the application of the satisfaction of the predicates.

As discussed in Section 4.3, the detection is driven by the messages (containing the local contexts) sent from context collecting processes. When receiving the local contexts, the property detection process on the property detection server first maintains the STD or the LAT according to different requirements of the predicate detection algorithms, and then detects the specified predicate [12], [28], [31]. When the predicate is detected true, the property detection process will notify the application (through the broker).

The maintenance of the STD or the LAT is generally reusable and independent to the detection. Thus, this layer should support the maintenance of the STD or the LAT, respectively. Like the design of context collecting processes in Section 5.4.1, we also adopt the template method pattern [41] to encapsulate different predicate detection algorithms and to facilitate the development of new property detection processes. We adopt a checker factory to manage the creation of property detection processes.

5.4.1 Property Detection over the STD

The abstract class AbstractFIFOChecker achieves the maintenance of the STD. Property detection processes which rely on the STD (e.g., [25], [26], [42], [12]), only need to extend the abstract class and implement the specific detection algorithms in the method handle(), as shown in Fig. 13.

The AbstractFIFOChecker adopts message buffers (the FIFOMsgBuffers) to ensure that messages from the same context collecting process are FIFO. The maintenance of the STD is transparent to the detection algorithms, which relieves the detection algorithms from the processing of message orders. When a new message is received, the method handle() will be invoked automatically to detect whether the new local context makes the predicate true. Detailed discussions on specific predicate detection algorithms over the STD can be found in [12], [12].
As for property detection over the LAT, the abstract class `AbstractApplication` and the interface `ResultCallback`.

**5.4.2 Property Detection over the LAT**

As for property detection over the LAT, the abstract class `AbstractLatticeChecker`, which is a subclass of the `AbstractFIFOChecker`, achieves the maintenance of the LAT. Property detection processes which rely on the LAT (e.g., [27], [28], [31]) only need to extend the abstract class and implement the specific detection algorithms in the method `check()`, as shown in Fig. 14.

The `AbstractLatticeChecker` constructs the LAT as a concrete subclass of the abstract class `AbstractLattice` in Fig. 13. Different detection algorithms (e.g., [28], [31]) may store different historical data on the LAT nodes to achieve incremental detection, as required in Section 4.3. Thus, we provide the abstract class `AbstractLatticeNode` for different detection algorithms to realize different types of LAT nodes. Furthermore, different LAT construction algorithms (e.g., [28], [30]) are supported by implementing the abstract method `expandLattice()` of the `AbstractLattice`. The construction of the LAT is transparent to the detection algorithms, which shields the detection algorithms from the burden of maintaining the LAT.

When a new local context arrives, the `AbstractLatticeChecker` first updates the LAT incrementally, and then invokes the method `check()` to detect whether the new part of the LAT makes the predicate true. Detailed discussions on specific LAT maintenance algorithms and predicate detection algorithms over the LAT can be found in [28], [31], [30].

**6 Simplifying Application Development**

In this section we discuss how the PD-CA framework simplifies the development of context-aware applications.

**6.1 Application Development**

Based on MIPA, context-aware applications specify contextual properties to the middleware and conduct the corresponding context-aware behavior when the properties are satisfied. The context-aware adaptation logic of the application is constructed in a condition-action manner. A programming toolkit is provided to facilitate the development of context-aware applications. Specifically, an abstract class `AbstractApplication` is adopted to encapsulate the interactions between the applications and the middleware, and a callback interface `ResultCallback` is introduced for the middleware to notify the applications to conduct context-aware behavior, as shown in Fig. 14. Concrete applications only have to extend the class `AbstractApplication`, specify contextual properties in XML as discussed in Section 5.2.1 and implement the corresponding context-aware behavior in the method `callback()` of the `ResultCallback`.

Applications register/unregister contextual properties and the corresponding callbacks to the middleware through the method `register()`/`unregister()`. When a contextual property is detected true, the middleware will trigger the corresponding callback of the application, which conducts the context-aware behavior.

**6.2 Exemplar Application**

The best demonstration of the middleware’s ability to ease context-aware application development is by examples. We exemplify the development of context-aware applications using the chemical plant safety management application in Section 2.1.

Assume that the robots patrol the plant from 8:00 to 18:00. At 8:00, the application starts the robots and registers the contextual properties (e.g., \(\phi_1\) in Section 2.1) and the corresponding callbacks to the middleware. When the contextual properties are detected true, the corresponding context-aware behavior will be conducted. At 18:00, the application unregisters the contextual properties and stops the robots. The code of the application is shown in Fig. 15.
We take the contextual property $\phi_1$ (i.e., all the robots detect hazardous material leak in a workshop) in Section 2.1 as an example. When $\phi_1$ is detected, the safety management application should sound the alarm, notify the workers in the workshop to escape, and notify the plant safety officer nearby to cope with the leak. Property $\phi_1$ in XML is in Fig. 8 and the corresponding context-aware behavior is implemented in the class OnLeak in Fig. 15. As shown in Fig. 15, the application registers the property (config/predicate/onLeak.xml) and the callback (onLeak) to the middleware. When the property is detected true, the method callback() of the callback onLeak will be invoked automatically to notify the workers and the plant safety officer.

The code of the application is reasonably clean and concise. Under the guidance of our PD-CA framework, MIPA allows the application developers to focus on the development of context-aware behavior, while not having to explicitly cope with the distributed and asynchronous contexts.

7 Experimental Evaluations

In this section we evaluate the performance of MIPA. We vary the types of predicates, the number of predicates, and the number of context collecting devices to study the scalability of MIPA. The detection of snapshot predicates and sequence predicates with respect to different degrees of asynchrony in the computing environment has been studied in our previous work [12], [28], [31]. Please refer to our previous work for detailed evaluations of specific detection algorithms.

7.1 Experiment Design

We implement MIPA with Java SE 1.6 and run MIPA over JVM 1.6 on a PC with Windows 7 (x64), an Intel Core i5-2400 Quad-Core Processor (3.10GHz), and 8GB RAM. We simulate the plant safety management scenario discussed in Section 2.1 and 6.2.

We conduct the experiments with two representative predicates: conjunctive predicate $\phi_1$ and regular expression predicate $\phi_3$ in Section 2.1. We tune the number of predicates and the number of robots, to study the memory consumptions of the property detection server and the context collecting clients, and the response latency of the property detection server. We use $M_s$ to denote the average of the memory consumption of the property detection server (all the context collecting clients on the robots) during the lifetime of the experiment, and use $T_s$ to denote the average time from the instant when the property detection process is triggered to the instant when the detection finishes.

The application registers a predicate to MIPA every 1 s. The robots sense context data every 400 ms. We simulate the context data of the robots with the Poisson distribution. The average time of the local activities (where the local predicate is true) on the robots is 5 mins, and the average interval between the activities (where the local predicate is false) is 1 min. The context data of each robot is up to 16,000. We model the communication delay between the robots by exponential distribution with average delay 10 ms. The lifetime of the experiment is up to 2 hours.

7.2 Evaluation Results

7.2.1 Performance with Conjunctive Predicates

We first evaluate the performance of MIPA with conjunctive predicate $\phi_1$ in Section 2.1. We tune the number of conjunctive predicates from 1 to 1,000 and the number of robots from 10 to 40.

As shown in Fig. 16, when fixing the number of robots, $M_s$ increases linearly with the number of predicates. When fixing the number of predicates, $M_s$ increases slowly with the number of robots. The slow increase of $M_s$ is because the detection algorithm for conjunctive predicates is space-efficient [12]. When the number of robots is 10, 20, 30, and 40, $M_s$ of 1,000 predicates is 49.5 MB, 89.8 MB, 148.1 MB, and 225.4 MB, respectively. Thus, $M_s$ is very small when detecting conjunctive predicates.

As shown in Fig. 17, when fixing the number of robots, $M_s$ increases linearly with the number of predicates. When fixing the number of predicates, $M_s$ increases slowly with the number of robots. When the number of robots is 10, 20, 30, and 40, $M_s$ of 1,000 predicates is 49.5 MB, 89.8 MB, 148.1 MB, and 303.7 MB, respectively. Thus, $M_s$ is acceptable on resource-constrained context collecting devices, such as mobile robots and phones.

As shown in Fig. 18, when fixing the number of robots, $T_s$ remains almost the same. When fixing the

![Fig. 16. The average memory consumption of the property detection server under conjunctive predicates.](image-url)
number of predicates, \( T_s \) increases slowly with the number of robots. When we tune the number of predicates from 1 to 1,000, \( T_s \) remains pretty small (within 30 \( \mu s \)). The reason is that the detection algorithm for conjunctive predicates is time-efficient [12].

### 7.2.2 Performance with Regular Expression Predicates

We then evaluate the performance of MIPA with regular expression predicate \( \phi_4 \) in Section 2.1. We tune the number of regular expression predicates from 1 to 100 and the number of robots from 3 to 6.

As shown in Fig. 19, when fixing the number of robots, \( M_c \) increases almost linearly with the number of predicates. When fixing the number of predicates, \( M_c \) increases exponentially with the number of robots. This is in accordance with the exponential detection algorithm for regular expression predicates [28]. When the number of robots is 3, 4, 5, and 6, \( M_c \) of 100 predicates is 2.8 MB, 3.2 MB, 3.4 MB, and 3.7 MB, respectively. \( M_c \) increases almost linearly with the number of robots, and costs much less than \( M_s \). This is because, although the detection algorithm for regular expression predicates on the property detection process is exponential (as shown in Fig. 19), the algorithm on the context collecting process is linear [28]. \( M_c \) is acceptable on resource-constrained context collecting devices.

As shown in Fig. 21, when fixing the number of robots, \( T_s \) increases linearly with the number of predicates. When the number of robots is 3, 4, and 5, \( T_s \) of 100 predicates is 0.2 ms, 1.9 ms, 24.5 ms,
respective. When the number of robots is 6, $T_s$ of 100 predicates increases quickly to 616.4 ms. $T_s$ increases exponentially with the number of robots. This is in accordance with the exponential detection algorithm for regular expression predicates \cite{28}. Thus, the detection of regular expression predicates can only work well in a small scale of context collecting devices.

Although the performance with respect to regular expression predicates is not as scalable as that with respect to conjunctive predicates, in pervasive computing scenarios, the number of context collecting devices is usually on a small scale \cite{28}. Thus, the performance of detection of regular expression predicates is often acceptable.

8 RELATED WORK

Context-aware computing has been extensively studied in the literature. From the point of view of this work, we mainly discuss two types of related work: the key enabling techniques and the software engineering methodologies for context-aware computing.

As for the key enabling techniques, tuple-space-based approaches provide an egocentric view of the dynamic environment for an individual application. Contexts are abstracted as data items stored in a distributed tuple space. The application achieves context-awareness by interacting with the contexts in the tuple space \cite{43}, \cite{44}, \cite{45}. LIME enables mobile coordination by abstracting communication into the tuple space \cite{43}. EgoSpaces enables each agent to specify its egocentric view of the network \cite{44}. TOTA, augmenting tuple spaces with reactive capabilities, provides a push-based interaction mechanism, i.e., tuples are propagated from a reference node based on context properties in a manner similar to content-based multicast \cite{45}. Our approach, instead of maintaining available contexts for the application, enables the application to declare its concern using high-level predicates. The middleware maintains contexts related to the predicates, conducts predicate detection, and notifies the application of the satisfaction of the predicates.

Query-based approaches provide a database-like abstraction of the computing environment of an application. The collection of contextual information available to a particular application is abstracted as a global virtual data repository that reflects the continuously changing state of the application’s environment. The application achieves context-awareness by querying the “database”. PerLa \cite{37} provides a SQL-like language which is able to manage heterogeneous devices and support most existing sampling modes. Payton et al. \cite{46} develop a self-assessing query processing protocol for dynamic environments, which not only delivers a query result but also labels that result with the achieved consistency semantics. Our approach also provides a specification for an application to express contextual properties of its interest. Though the specification of predicates in our PD-CA framework has less expressiveness compared to database query languages, our work mainly targets at coping with the asynchrony of the computing environment.

Pub/sub-based approaches, such as Solar \cite{47} and STEAM \cite{48}, enable a number of subscribers to continuously retrieve events from a number of publishers. These approaches develop efficient multicast-based routing and in-network event filtering techniques. Our predicate-detection-based approach is analogous to pub/sub-based approaches in that the application subscribes the predicates to the middleware, while the middleware subscribes the consisting local predicates to the distributed context collecting devices. Our work is a special case of the general pub/sub approach, but it is dedicated to coping with the asynchrony of the computing environment.

Recently, many software methodologies for the development of context-aware applications are proposed. Anind K. Dey \cite{5} identifies a design process for building context-aware applications and provides two programming abstractions to facilitate the design of such applications. The Context Toolkit is built to support the design process and the programming abstractions. Henricksen et al. \cite{49} introduce a graphical context modelling approach, a preference model for representing context-dependent requirements, and two programming models for developing context-aware applications. A generic software engineering process is also proposed when building context-aware applications using these tools. Cassou et al. \cite{50} introduce a design methodology as well as corresponding tool support for pervasive applications. Testing and maintenance are also discussed.

We propose the PD-CA framework to enable context-awareness in asynchronous pervasive computing environments enriched with C³S devices. The PD-CA framework employs the predicate detection
theory to address the challenge of the asynchrony. PD-CA also includes a design process for developing applications aware of asynchronous environments.

9 CONCLUSION AND FUTURE WORK

In this paper, we study how to enable context-awareness in asynchronous pervasive computing environments enriched with C/S devices. We propose the PD-CA framework, which utilizes the predicate detection theory to cope with the intrinsic asynchrony of the pervasive computing environment. PD-CA also proposes a design process for developing context-aware applications. Under the guidance of PD-CA, we design and implement the MIPA middleware. We also demonstrate how MIPA simplifies the development of context-aware applications.

This work is being expanded in various directions. To better work in real scenarios, fault-tolerance in the PD-CA framework must be strengthened. The most urgent issues to be addressed include the crash failure of context collecting devices and the link failure of wireless communications. The PD-CA framework also needs to support multiple time models in the future. For example, the partially synchronous model should be integrated in the PD-CA framework to cover more pervasive computing scenarios.

ACKNOWLEDGEMENTS

This work is supported by the National Natural Science Foundation of China (No. 61272047, 91318301).

REFERENCES

[1] G. Zhan and W. Shi, “Lobot: Low-cost, self-contained localization of small-sized ground robotic vehicles,” IEEE Trans. Parallel Distrib. Syst., vol. 24, no. 4, pp. 744–753, 2013.
[2] P. S. Duggirala, T. Johnson, A. Zimmerman, and S. Mitra, “Static and dynamic analysis of timed distributed traces,” in Proc. of IEEE Real-Time Syst. Symp. (RTSS’12), 2012.
[3] N. D. Lane, E. Miluzzo, H. Lu, D. Peebles, T. Choudhury, and A. T. Campbell, “A survey of mobile phone sensing,” IEEE Communications Magazine, vol. 48, no. 9, pp. 140–150, 2010.
[4] M. Azizyan, I. Constandache, and R. Roy Choudhury, “Surroundsense: Mobile phone localization via ambience fingerprinting,” in Proc. Annual Intl. Conf. on Mobile computing and networking (MobiCom’09), 2009, pp. 261–272.
[5] A. Dey, “Providing architectural support for building context-aware applications,” Ph.D. Thesis, Georgia Institute of Technology, Nov 2000.
[6] 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.
[7] 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.
[8] A. D. Kshemkalyani and J. Cao, “Predicate detection in asynchronous pervasive environments,” IEEE Trans. on Computers, vol. 62, no. 9, pp. 1823–1836, 2013.
[9] F. B. Schneider, “What good are models and what models are good?” Distrib. Syst. (2nd Ed.), pp. 17–26, 1993.
[10] B. Patt-Shamir and S. Rajsbaum, “A theory of clock synchronization,” in Proc. Annual ACM Symp. on Theory of Computing (STOC’94), 1994, pp. 810–819.
[35] O. Babaoğlu and M. Raynal, “Specification and verification of dynamic properties in distributed computations,” *J. Parallel Distrib. Comput.*, vol. 28, no. 2, pp. 173–185, 1995.

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

[37] A. Schreiber, R. Camplani, M. Fortunato, M. Marelli, and G. Rota, “PerLa: A language and middleware architecture for data management and integration in pervasive information systems,” *IEEE Trans. Softw. Eng.*, vol. 38, pp. 478–496, 2012.

[38] K. Aberer, M. Hauswirth, and A. Salehi, “A middleware for fast and flexible sensor network deployment,” in *Proc. Intl. Conf. on Very Large Data Bases (VLDB’06)*, 2006, pp. 1199–1202.

[39] N. W. Paton and O. Diaz, “Active database systems,” *ACM Comput. Surv.*, vol. 31, no. 1, pp. 63–103, Mar 1999.

[40] A. T. S. Chan and S.-N. Chuang, “MobiPADS: A reflective middleware for context-aware mobile computing,” *IEEE Trans. Softw. Eng.*, vol. 29, no. 12, pp. 1072–1085, 2003.

[41] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[42] 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.

[43] A. L. Murphy, G. P. Picco, and G.-C. Roman, “LIME: A coordination model and middleware supporting mobility of hosts and agents,” *ACM Trans. Softw. Eng. Methodol.*, vol. 15, no. 3, pp. 279–328, 2006.

[44] C. Julien and G.-C. Roman, “EgoSpaces: Facilitating rapid development of context-aware mobile applications,” *IEEE Trans. Softw. Eng.*, vol. 32, no. 5, pp. 281–298, May 2006.

[45] M. Mamei and F. Zambonelli, “Programming pervasive and mobile computing applications: The TOTA approach,” *ACM Trans. Softw. Eng. Methodol.*, vol. 18, pp. 15:1–15:56, Jul 2009.

[46] J. Payton, C. Julien, G.-C. Roman, and V. Rajamani, “Semantic self-assessment of query results in dynamic environments,” *ACM Trans. Softw. Eng. Methodol.*, vol. 19, no. 4, pp. 12:1–12:33, Apr 2010.

[47] G. Chen, M. Li, and D. Kotz, “Data-centric middleware for context-aware pervasive computing,” *Pervasive and Mobile Computing*, vol. 4, no. 2, pp. 216–253, 2008.

[48] R. Meier and V. Cahill, “On event-based middleware for location-aware mobile applications,” *IEEE Trans. Softw. Eng.*, vol. 36, no. 3, pp. 409–430, 2010.

[49] 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.

[50] D. Cassou, J. Bruneau, C. Consel, and E. Balland, “Toward a tool-based development methodology for pervasive computing applications,” *IEEE Trans. Softw. Eng.*, vol. 38, no. 6, pp. 1445–1463, 2012.