Enabling Context-Awareness by Predicate Detection in Asynchronous Environments

Yiling Yang, Yu Huang, *Member, IEEE*, Xiaoxing Ma, *Member, IEEE*, 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 of the computing environment. Predicate detection is a well studied technique dedicated to detecting global predicates over asynchronous computations and can be employed to achieve context-awareness of the asynchronous environment. However, there is no methodological framework which guides us to systematically apply the abstract predicate detection theory to the development of concrete context-aware applications. To this end, we present the Predicate Detection-based Context-Awareness (PD-CA) framework. PD-CA maps the concepts of context-awareness to concepts of predicate detection. PD-CA also presents a design process of providing middleware support for context-aware applications. Under the guidance of the PD-CA framework, we design and implement the Middleware Infrastructure for Predicate detection in Asynchronous environments (MIPA). We also propose the programming toolkit to facilitate the development of context-aware applications based on MIPA, and demonstrate the use of the toolkit by a case study of a chemical plant safety management application. Experimental evaluations show the performance of MIPA in enabling context-awareness despite of the asynchrony.

Index Terms—Context-awareness, predicate detection, asynchrony, context-aware middleware

1 Introduction

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

These mobile devices can provide rich context information for the applications [3], [4], and pervasive applications are typically designed to be context-aware, i.e., intelligently adapting their behavior to the environment [5], [6]. However, enabling context-awareness over these mobile 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 device. Thus, a group of autonomous but also coordinating mobile 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.

- *Y. Yang, Y. Huang, X. Ma, and J. Lu are with the State Key Laboratory for Novel Software Technology and the Institute of Computer Software, Nanjing University, Nanjing, China 210023. E-mail: csylyang@gmail.com, {yuhuang, xxm, lj}@nju.edu.cn.*

Manuscript received 4 Apr. 2014; revised 12 Apr. 2015; accepted 14 Apr. 2015. Date of publication 20 Apr. 2015; date of current version 15 Jan. 2016. Recommended for acceptance by P. Bellavista.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2015.2424879

The coordination among the devices is intrinsically asynchronous. There is no global clock available among the devices. Constrained resources and task scheduling of the mobile devices (often embedded systems) may lead to unpredictable computation delay [7]. The growing adoption of wireless communications, which are prone to bandwidth shortage, network congestion, unpredictable routings, and retransmission, leads to unpredictable communication delay [8], [9]. All these characteristics of the devices and their communication networks lead to the intrinsic asynchrony among the contexts they collect [7], [9].

The asynchrony among contexts may greatly affect how the application interprets the collected contexts, thus leading to wrong adaptations. For example, when all the three patrolling robots detect that "I sense no people around", we cannot take it for granted that there are actually no people around and trigger an alarm to call for people inspection. Only when we make sure that these three pieces of contexts belong to the same snapshot of time, can we decide that no one is in the room. Also the application may be concerned of the temporal order among contextual activities (e.g., "some robot detects light of fire and then all robots detect temperature increase"). We need to explicitly build the temporal relation among contexts, which is also greatly affected by the asynchrony among contexts.

One possible solution to coping with the asynchrony is clock synchronization. However, clock synchronization may not enable correct and fault-tolerant coordination among the autonomous mobile devices [2], [8]. Thus, it cannot enable context-awareness despite of the asynchrony in pervasive scenarios enriched with coordinating mobile devices. Specifically, each device only has its own local clock, which cannot be perfectly synchronized [2], [10]. The uncertainty caused by

the skew among the clocks may lead to incorrect behavior [7]. Besides, clock synchronization schemes make assumptions on process execution speeds and communication delay. These assumptions may not be guaranteed for the autonomous devices. A group of robots are prone to incorrect behavior even if one single assumption is violated [8]. The inaccuracy of synchronization and the potential violation of assumptions make reasoning based on time and timeouts a delicate and error-prone undertaking [8]. 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 [7], [9]. Consequently, it is more practical to have few or, better, no synchrony assumption in scenarios of coordinating mobile devices [2], [7], [8].

Predicate detection is a well studied technique dedicated to detecting global predicates over asynchronous computations [9], [11], [12], [13]. Based on predicate detection, global properties of the underlying asynchronous system can be understood at runtime. Context-awareness in asynchronous environments can be achieved based on predicate detection in principle, by runtime detection of properties concerning the environment (both the physical environment and the status of the coordination among the devices) specified by context-aware applications over asynchronous contexts. However, there is no methodological framework which guides us to enable context-awareness based on predicate detection concepts and techniques in a systematic way.

To address the challenges above, we propose the Predicate Detection-based Context-Awareness (PD-CA) framework. PD-CA maps the concepts in context-aware computing to concepts in predicate detection. PD-CA also presents a design process to provide middleware support for developing context-aware applications. Under the guidance of the PD-CA framework, we design and implement the Middleware Infrastructure for Predicate detection in Asynchronous environments (MIPA). We also present the programming toolkit for application development based on MIPA, and demonstrate the use of the toolkit by a case study of a chemical plant safety management application. Experimental evaluations show the performance of MIPA in enabling context-awareness despite of the asynchrony.

The rest of this work is organized as follows. Section 2 presents the PD-CA framework. Section 3 discusses the design and implementation of MIPA. Section 4 discusses the programming toolkit. Section 5 presents the experimental evaluations. Section 6 discusses the related work. In Section 7, we conclude this work and discuss the future work.

2 The PD-CA Framework

In this section, we first introduce the essential concepts in predicate detection. Then we present the concept mapping from context-aware computing to predicate detection. Finally we propose the design process of providing middleware support for context-aware applications under the PD-CA framework.

2.1 Essential Concepts in Predicate Detection

Predicate detection refers to the checking of global predicates over asynchronous computations [11], [12], [14]. Predicate detection consists of three essential parts [9], [13]: 1) modeling of the system execution trace; 2) specification of global predicates; 3) detection of specified predicates.

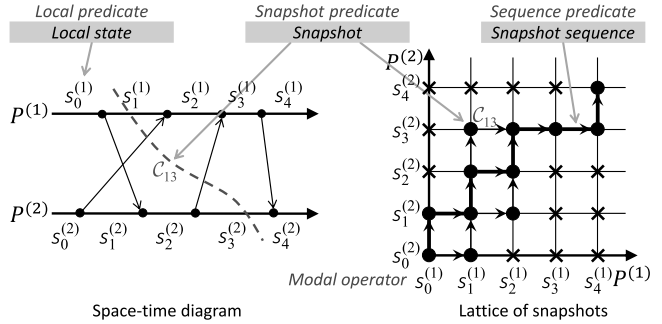


Fig. 1. The space-time diagram and its corresponding lattice of snapshots. The lattice contains all possible snapshots (i.e., the black dots), and the black broken line is one possible sequence of snapshots.

In the modeling of the system execution trace, we rely on the asynchronous message-passing model. Without synchronized global clock, *logical time* is used to cope with the asynchrony among the execution trace generated by the asynchronous system of *instrumented processes*. Temporal orders among events of the instrumented processes are captured by the *happen-before* relation resulting from message causality [15], and encoded/decoded via the logical vector clock [16]. A *global snapshot* of the asynchronous system, i.e., a vector of pairwise concurrent *local states* of each process, is redefined under the notion of logical time. Dynamic behavior of the system is modeled over sequences of global snapshots derived from the lattice structure (denoted by *LAT*) of global snapshots, as shown in Fig. 1. Due to the uncertainty caused by the asynchrony, we can obtain multiple totally-ordered sequences of snapshots from *LAT*. We only know that the actual execution of the system is one of these sequences but never know which one [13].

In the specification of global predicates, we can specify local predicates (e.g., in first-order logic) on local states of one instrumented process, global predicates (e.g., *conjunctive predicates* [9]) on snapshots, and sequence predicates (e.g., *regular expression predicates* [13] and *CTL predicates* [17]) on sequences of snapshots, as shown in Fig. 1. Modal operators $Pos(\cdot)$ and $Def(\cdot)$ are adopted to cope with the uncertainty of multiple possible sequences of snapshots derived from *LAT* [11]. Informally, $Pos(\phi)$ means that the predicate ϕ holds on at least one possible sequence of snapshots, and $Def(\phi)$ means that ϕ holds on all possible sequences.

In the detection of specified predicates, a centralized *checker process* is in charge of collecting the execution trace of the system and detecting the specified predicate over the observed multiple possible sequences of snapshots. Notice that a general algorithm to detect all types of predicates is usually prohibitively expensive and impractical [11]. Thus, specific efficient algorithms are designed for different types of predicates over the Space-Time Diagram (denoted by *STD*) or the lattice *LAT* [9], [13], [17].

Please refer to our previous work for more detailed discussions on predicate detection [9], [13], [17].

2.2 The Concept Mapping

To enable context-awareness by predicate detection, we need to map concepts of context-awareness to that of

TABLE 1
The Concept Mapping between Predicate Detection and Context-Aware Computing

	Concepts in PD	Concepts in CA
Modeling	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
Specification	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
Detection	checker process	property detection process on the middleware
	detection of predicates	persistent monitoring of specified contextual properties

predicate detection. The concept mapping consists of three essential parts in accordance with the three parts of predicate detection, as outlined in Table 1.

2.2.1 Modeling of Asynchronous Contexts

The key problem in the modeling of asynchronous contexts is to cope with the intrinsic asynchrony in the pervasive environment. The collected contexts are *spatially distributed* and *temporally asynchronous*. We discuss these two characteristics in detail below.

A context-aware application is often concerned with *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 devices. For example, for property ϕ_1 (i.e., "*all the robots detect no people in a workshop*"), robot R_1 can provide the type of local context "R_1 *detects no people around*". Based on the predicate detection theory, the *type of local context* provided by a device is modeled as a local variable on the device. The value of local context (e.g., "R_1 *detects no people around*" is true) and its logical timestamp define a *local state* of the device. Changes of local contexts are modeled as *contextual events*. Based on the notion of global snapshot, we model the spatially distributed contexts by a *hierarchy of global-local contexts*. A vector of local contexts from each device forms a *global context* (e.g., "*all the robots detect no people in a workshop*"). This is in accordance with our intuitive understanding towards the computing environment.

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 vector of local contexts from the devices forms a *meaningful global context* of the computing environment. A meaningful global context means it can be possibly observed in the actual execution of the system of context collecting devices. Based on the predicate detection theory, we rely on logical time which is maintained over message exchanges between the devices, to model the temporal order among the local contexts of different devices. Based on the notion of global snapshot, a vector of local contexts over the local states of a global snapshot of the system, forms a meaningful global context. As the context collecting devices persistently collect local contexts, we can get sequences of meaningful global contexts over sequences of global snapshots of the system, which represent the temporal evolution of the environment. As we can get multiple possible sequences of snapshots from *LAT*, we can get multiple possible temporal evolutions of the environment but we cannot tell which is the actual one.

2.2.2 Specification of Contextual Properties

In our PD-CA framework, the context-aware application states contextual properties of the environment by formal specification of global predicates. The specification is in the form of a *hierarchy of predicates* defined on the hierarchy of global-local contexts discussed in Section 2.2.1. We introduce the hierarchy of predicates in a bottom-up manner.

- *Local predicates*. Local predicates are specified over local contexts of context collecting devices, to indicate the application's concerns on local properties of the computing environment. For example, $LP_i =$ "R_i *detects no people around*" in property ϕ_1 is a local predicate. We currently support local predicates in first-order logic composed of atomic predicates defined on local contexts.
- *Snapshot predicates*. Snapshot predicates are specified over meaningful global contexts, to indicate the application's concerns on global instantaneous properties of the environment. We currently support conjunctive predicates in the form of conjunction of local predicates [9]. For example, the property ϕ_1 can be expressed as a snapshot predicate $\phi_1 =$ "$LP_1 \wedge LP_2 \wedge LP_3$".
- *Sequence predicates*. The application can specify sequence predicates over sequences of meaningful global contexts to describe dynamic properties and behavioral patterns of the environment. Various types of sequence predicates can be specified based on the predicate detection theory. We currently support regular expression predicates and CTL predicates defined over a set of snapshot predicates [13], [17]. For example, the property ϕ_2 concerning the behavior of the robots "*the robots pass the gateway between two workshops in certain order*" can be expressed by a regular expression predicate $\phi_2 =$ "$a^*ab^*bc^*cd^*d$" with 'a' indicating that R_1, R_2, and R_3 are all in workshop A, 'b' indicating that R_2 and R_3 are in workshop A while R_1 is in workshop B, 'c' indicating that R_3 is in workshop A while R_1 and R_2 are in workshop B, and 'd' indicating that R_1, R_2, and R_3 are all in workshop B. More examples can be found in [13], [17].
- *Modal operators*. Modal operators $Pos(\cdot)$ and $Def(\cdot)$ are adopted to interpret the predicates over the multiple possible sequences of meaningful global contexts. $Pos(\cdot)$ is often used to eliminate the possibility of "bad things", while $Def(\cdot)$ is often used to ensure the occurrence of "good things". For example, we adopt the modal operator $Pos(\cdot)$ for the snapshot predicate ϕ_1, i.e., $Pos(\phi_1)$, since we are concerned of even the possibility of such a bad thing.

Note that our specification is defined over the asynchronous message-passing model as in the predicate

detection theory, we can only express predicates concerning the order relations but cannot express predicates with real-time constraints.

2.2.3 Detection of Specified Properties

In order to employ the predicate detection algorithms to achieve context-awareness, a *context collecting process* (counterpart of the instrumented process in predicate detection) is deployed on each of the coordinating context collecting devices which can provide contexts related to the specified property, and the checker process in predicate detection is instantiated as a *property detection process* deployed on the middleware. The context collecting processes are in charge of detecting local predicates, maintaining logical time based on message exchanges, and pushing local states (i.e., local contexts with logical timestamps) to the property detection process. The property detection process detects the specified predicates with the corresponding predicate detection algorithms 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 is achieved in an *event-driven* manner. When a context collecting device collects new local raw contexts, the context collecting process on the device will first filter the raw contexts by the subscribed local predicate, and send local states (when the value of the local predicate changes) to the property detection process. The property detection process is then triggered to detect whether the newly arrived local contexts make the specified predicate true.

Though specific predicate detection algorithms are designed for different types of predicates according to the predicate detection theory, predicate detection in our PD-CA framework is achieved in two steps in principle. The property detection process first maintains *STD* or *LAT* according to different requirements of specific algorithms, and then invokes the corresponding algorithm to detect the specified predicate over *STD* or *LAT*. Note that the maintenance of *STD* or *LAT* is generally reusable and independent with the detection algorithms for specific types of predicates. Implementation details of the predicate detection algorithms will be discussed in Section 3.2.2.

2.3 Design Process of PD-CA

In this section, we discuss the design process of providing middleware support for context-aware applications using the PD-CA framework. We illustrate the design process with the chemical plant scenario introduced in Section 1, as shown in Fig. 2. The design process includes five steps, as detailed below.

Step 1: Obtainment of the specification. The application specifies the contextual property (e.g, property ϕ_1) to the middleware. The middleware then parses the property, identifies the consisting local predicates in the property, registers the local predicates to their corresponding context collecting devices (as shown in Fig. 2), and launches a property detection process on the middleware to detect the contextual property.

Step 2: Acquisition of asynchronous contexts. Each context collecting device registers itself to the middleware with the type of local context it provides. When the context collecting device receives the registration of a local predicate, a context collecting process is launched on the device. The context collecting process filters collected local raw contexts with the local predicate (e.g., robot R_1 detects local predicate LP_1). It also maintains the logical time by piggybacking logical timestamps on messages between the devices [9], [13]. When the context collecting process generates a new local state, it pushes the local state to the property detection process on the middleware.

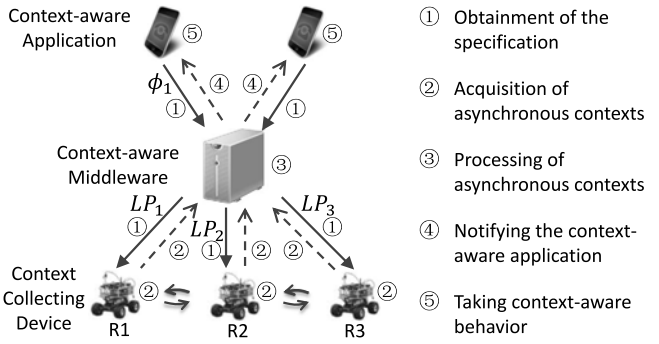


Fig. 2. The design process in the chemical plant scenario.

Step 3: Processing of asynchronous contexts. When the property detection process on the middleware receives the local contexts from distributed context collecting processes, it first explicitly maintains *STD* or *LAT* based on the logical timestamps of these asynchronous contexts, and then detects the specified contextual property by the corresponding predicate detection algorithm.

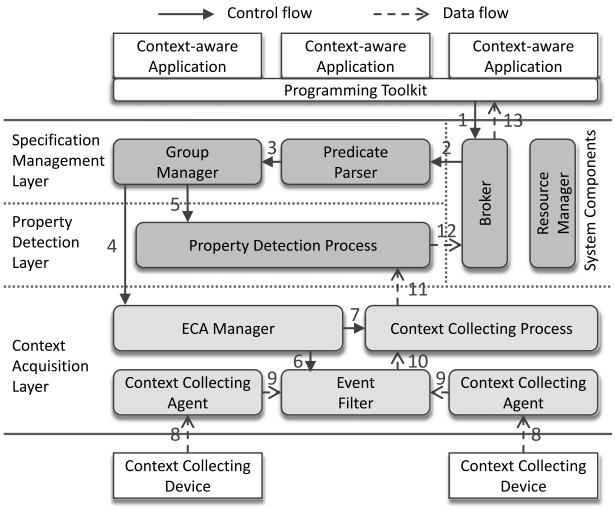
Step 4: Notifying the context-aware application. If the contextual property is detected true, the middleware notifies the context-aware application.

Step 5: Taking context-aware behavior. When notified of the satisfaction of the property, the application conducts the corresponding context-aware behavior (e.g., when ϕ_1 is true, the application will sound the alarm in the workshop and notify the plant manager of the absence).

The design process guides the architecture design of the context-aware middleware, as presented in the following Section 3. With the middleware support, the application developers can be relieved from the time- and energy-consuming burdens of collecting asynchronous contexts from distributed heterogeneous context collecting devices and processing these asynchronous contexts. Detailed discussions can be found in Section 4.

3 MIPA - Providing Middleware Support for Predicate Detection-Based Context-Awareness

In this section, we present the design and implementation of MIPA—*Middleware Infrastructure for Predicate detection in Asynchronous environments*, which greatly simplifies the tasks of building context-aware applications under the guidance of the PD-CA framework [18], [19]. We first discuss the overall architecture, then introduce the support for the concept mapping, and finally discuss the support for the design process.



1. Specifying a contextual predicate
2. Transferring the predicate to the parser
3. Starting a group of processes
4. Registering the local predicate to the CC-client
5. Creating a property detection process
6. Setting the local predicate as the event condition
7. Creating a context collecting process
8. Collecting local raw contexts
9. Pushing raw contexts to the event filter
10. Sending filtered local contexts
11. Sending local contexts to the property detection process
12. Notifying the broker of the satisfaction of the predicate
13. Notifying the application of the satisfaction of the predicate

Fig. 3. The architecture and main components of MIPA.

3.1 Overview of MIPA Architecture

The design process in the PD-CA framework (introduced in Section 2.3) 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. 3:

- The *specification management layer* is in charge of parsing the contextual property from the application, and maintaining the group of processes involved in the detection of the property, as in Step 1 of the design process.
- The *context acquisition layer* is in charge of collecting asynchronous contexts on context collecting devices, and pushing them to the property detection server, as in Step 2 of the design process.
- The *property detection layer* is in charge of detecting the specified property over the asynchronous contexts sent from the context collecting devices. When the property is detected true, the property detection layer will notify the application, as in Step 3-4 of the design process.
- The middleware also contains some *system components*. Specifically, the *broker* is in charge of interacting with the application and remote context collecting clients, and delivering their requests to the corresponding components. The *resource manager* is in charge of managing the available types of local contexts and their corresponding providers.

The middleware can be deployed as a centralized *property detection server* (*PD-server*, for short), and several *context collecting clients* (*CC-clients*, for short) over context collecting devices. In Fig. 3, the components in dark gray color are running on the PD-server, and the components in light gray color are running on the CC-clients.

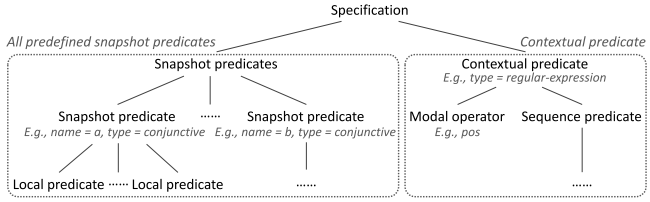


Fig. 4. The tree structure of the specification.

3.2 Support for the Concept Mapping

We first discuss the specification of contextual properties using global predicates in MIPA. Then we discuss the persistent monitoring of contextual properties using runtime predicate detection algorithms.

3.2.1 Specification Management

The specification serves as the contract between the application and MIPA. The predicates we support may have different types, such as conjunctive predicates or regular expression predicates. To facilitate the specification of predicates, we provide a uniform mechanism for the application to specify different types of predicates. Notice that sequence predicates are generally composed of a set of snapshot predicates, we extract these snapshot predicates as an alphabet to facilitate the description of sequence predicates. As 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. 4.

Specifically, the "*Snapshot predicates*" node contains all the snapshot predicates (i.e., the alphabet), and each "*Snapshot predicate*" node defines a snapshot predicate. The "*Contextual predicate*" node contains the specified predicate which is composed of snapshot predicates defined in the "*Snapshot predicate*" nodes.[1] Each "*Snapshot predicate*" node is labeled with a name and its type, e.g., "*conjunctive*" for conjunctive predicates. The "*Contextual predicate*" node is also labeled with its type, e.g., "*regular-expression*" for regular expression predicates. This structure can unify different types of predicates, and can be easily extended to include new types of snapshot and 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. As an example, property ϕ_1 (defined in Section 2.2.2) in XML is shown in Fig. 5. The DTD of the XML for specification can be found in Fig. 2 of the Appendix in the supplementary file, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TC.2015.2424879 available online.

When receiving the specification in XML from the application, a *predicate parser* is employed to parse the specification into the structure shown in Fig. 4. Notice that different types of snapshot and sequence predicates may have different structures. Thus, we provide different parsers for

1. A snapshot predicate can be viewed as a sequence predicate with only one snapshot predicate.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE specification SYSTEM "specification.dtd">
<specification>
  <snapshotPredicates>
    <snapshotPredicate name="a" type="conjunctive">
      <localPredicate>
        <formula>                                        local predicate LP1
          <atom name="RFID_R1" operator="not-contain" value="tag_anyone"/>
        </formula>
      </localPredicate>
      <localPredicate>
        <formula>                                        local predicate LP2
          <atom name="RFID_R2" operator="not-contain" value="tag_anyone"/>
        </formula>
      </localPredicate>
      <localPredicate>
        <formula>                                        local predicate LP3
          <atom name="RFID_R3" operator="not-contain" value="tag_anyone"/>
        </formula>
      </localPredicate>
    </snapshotPredicate>
  </snapshotPredicates>
  <contextualPredicate type="WCP">
    <modalOperator value="pos"/>
    <sequencePredicate>
      <element>                                          snapshot predicate φ1
        <cgs name="a"/>
      </element>
    </sequencePredicate>
  </contextualPredicate>
</specification>
```

Fig. 5. Property ϕ_1 in XML.

different types of predicates. For example, when the predicate parser reaches a snapshotPredicate node labeled with "conjunctive" as shown in Fig. 5, the type is first recognized and the corresponding parser for conjunctive predicates is invoked to parse the node. Our design provides good reusability and extensibility. New types of predicates can be parsed by adding new parsers accordingly.

After parsing the specified predicate, the consisting local predicates are identified, e.g., the three local predicates in Fig. 5. Then, several context collecting processes which monitor the value changes of the local predicates are launched on the CC-clients. Meanwhile, a property detection process which detects the specified predicate is launched on the PD-server. These processes are distributed on different CC-clients and the PD-server. To better maintain them, we organize the processes created for a predicate as a *property detection group*, and adopt a *group manager* to coordinate the creation and destruction of the processes.

3.2.2 Contextual Property Detection

The detection of contextual properties includes the context collection by context collecting processes and the runtime detection by property detection processes.

A context collecting process running on a CC-client is in charge of taking actions to event changes from the local sensors filtered by the local predicate, including updating local logical time and sending local contexts (i.e., values of the local predicate) with logical timestamps to the property detection process on the PD-server. Details of event filtering on CC-clients by the ECA mechanism can be found in [19].

Notice that detection algorithms for different types of predicates may have different logics of context collecting processes [9], [13], [17], but they share the same skeleton. We adopt the *template method* pattern to facilitate the development of context collecting processes. New types of context collecting processes can be easily integrated by extending the abstract class AbstractNormalProcess.[2]

Runtime detection by the property detection process is driven by messages (containing local contexts and logical timestamps) sent from context collecting processes, as discussed in Section 2.2.3. When receiving a new message, the property detection process first maintains *STD* or *LAT* according to the requirements of different detection algorithms [9], [13], [17], and then detects the specified predicate by corresponding detection algorithms. As the design of context collecting processes, we adopt the *template method* pattern to facilitate the development of property detection processes. We provide two abstract classes which maintain *STD* and *LAT* for the detection algorithms. We adopt a *checker factory* to manage the creation of property detection processes.

Specifically, to shield the detection algorithms from the maintenance of *STD*, we provide an abstract class AbstractFIFOChecker to achieve the maintenance of *STD*.[3] The abstract class adopts message buffers to ensure that messages from the same context collecting process are FIFO (First-in-First-Out). The maintenance of *STD* is transparent to the detection algorithms, which relieves the detection algorithms from the processing of message orders. New property detection algorithms which rely on *STD* (e.g., [9], [12]) only need to extend the abstract class and implement the specific detection algorithms (in the method handle(), as shown in Fig. 4 of the Appendix, available in the online supplemental material). When a new message is received, the specific detection algorithm will be invoked automatically to detect whether the newly constructed *STD* makes the predicate true. Detailed discussions on specific property detection algorithms over *STD* can be found in [9].

As for the detection algorithms over *LAT*, the abstract class AbstractLatticeChecker, a subclass of AbstractFIFOChecker, achieves the maintenance of *LAT* based on the maintenance of *STD*.[4] The abstract class can be instantiated to construct *LAT* by different lattice maintenance algorithms according to the requirements of the detection algorithms, e.g., tree-based lattice [17], indexed lattice [13], windowed lattice [20], etc. Notice that different detection algorithms may store different historical data on lattice nodes to achieve incremental detection. Thus, we only maintain the lattice structure in the abstract class, for different algorithms to realize different types of lattice nodes. The maintenance of *LAT* is also transparent to the detection algorithms. New property detection algorithms which rely on *LAT* (e.g., [13], [17], [20]) only need to extend the abstract class and implement the specific detection algorithms (in the method check(), as shown in Fig. 4 of the Appendix, available in the online supplemental material). When a new message arrives, *LAT* will be maintained incrementally, and the specific detection algorithm will be invoked automatically to detect whether the newly constructed *LAT* makes the predicate true. Detailed discussions on specific lattice maintenance algorithms and property detection algorithms over *LAT* can be found in [13], [17], [20].

2. The definition of AbstractNormalProcess can be found in Fig. 3 of the Appendix, available in the online supplemental material.

3. The definition of AbstractFIFOChecker can be found in Fig. 4 of the Appendix, available in the online supplemental material.

4. The definition of AbstractLatticeChecker can be found in Fig. 4 of the Appendix, available in the online supplemental material.

3.3 Support for the Design Process

In this section, we instantiate the abstract design process of PD-CA by providing middleware support for the chemical plant scenario using MIPA. In this scenario, a group of mobile robots patrols the plant periodically to detect workers in the workshops, and a centralized server is deployed for the *PD-server*. Before the robots start to patrol, a *CC-client* is launched on each robot R_i. The *CC-client* registers the type of local context (i.e., "R_i *detects no people around*") to the *resource manager* on the *PD-server* by the *ECA manager*.[5]

Step 1: Obtainment of the specification. The safety management application specifies the contextual property $Pos(\phi_1)$ in XML (as shown in Fig. 5) to the *PD-server* through the *broker*. The *broker* transfers the XML to the *predicate parser*, which then parses the predicate, identifies the type of the predicate ("conjunctive" in this case) and the consisting local predicates (e.g., LP_1 = "RFID_R1 not − contain tag_anyone" as shown in Fig. 5), and requests the *group manager* to launch a group of processes dedicated for the detection of the predicate, as in Step 1-3 in Fig. 3.

The *group manager* first locates the corresponding robot of each local predicate (e.g., robot R_1 for LP_1) through the *resource manager*, and registers each local predicate to the *CC-client* running on the corresponding robot. According to the type of the predicate, the *group manager* also creates a *property detection process* (for conjunctive predicates in this scenario) by the *checker factory*, as in Step 4-5 in Fig. 3.

Step 2: Acquisition of asynchronous contexts. On each robot, when the *CC-client* receives the registration of a local predicate from the *group manager*, the *ECA manager* first registers the local predicate to the *event filter* as the *event condition* to filter local raw contexts from the *context collecting agents*. The *ECA manager* then starts a *context collecting process* according to the type of predicate, to monitor the value changes of the local predicate, as in Step 6-7 in Fig. 3.

The *context collecting agents* push local raw contexts to the *event filter*. When the value of the local predicate changes, the *event filter* sends the value change to the *context collecting process*. The *context collecting process* then generates a new local context with logical timestamp, and sends it to the *property detection process* on the *PD-server*, as in Step 8-11 in Fig. 3.

Step 3: Processing of asynchronous contexts. On receiving a new local context, the concrete *property detection process* on the *PD-server* maintains *STD* or *LAT* (*STD* in this scenario) incrementally based on the logical timestamps and detects the predicate by the detection algorithm developed for the type of the predicate (algorithm over *STD* for conjunctive predicates in this scenario).

Step 4: Notifying the context-aware application. If the predicate is detected true (e.g., all the robots detect no people in the workshop), the *property detection process* notifies the application through the *broker*, as in Step 12-13 in Fig. 3.

Step 5: Taking context-aware behavior. When notified of the satisfaction of the predicate, the application conducts the corresponding context-aware behavior (e.g., sound the alarm in the workshop and notify the plant manager of the absence in this scenario).

5. Detailed discussions on the registration of context collecting devices can be found in [19].

4 Simplifying Application Development

In this section, we first introduce the programming toolkit which facilitates the development of context-aware applications based on MIPA. Then we demonstrate the use of the toolkit by an exemplar application.

4.1 The Programming Toolkit

Without the middleware support from MIPA, the application developers may suffer from the heavy burden of coping with the asynchrony of the contexts. As discussed in Section 1, the application developers cannot simply assume that the contexts collected belong to the same snapshot of time, or be labelled with the global clock timestamp. Thus they not only have to obtain temporal relations among the asynchronous contexts, but also have to reason the global temporal properties over the asynchronous contexts. More importantly, the resulting implementation is often application-specific, strongly coupled with the application logic and not reusable.

Based on MIPA, the application developers only have to specify contextual properties to the middleware and implement the corresponding context-aware behavior. The burdens of coping with the asynchronous contexts are shielded by the middleware. Under the guidance of the PD-CA framework, the processing of asynchronous contexts by the middleware is systematic and loosely-coupled with the application logic.

To leverage the middleware support for coping with the asynchrony, we further design a programming toolkit to facilitate the development of context-aware applications based on MIPA. Using the toolkit, the application specifies contextual properties, implements context-aware behavior, and registers the properties with their corresponding behavior to MIPA in a condition-action manner. When a contextual property is detected true by MIPA, the corresponding behavior should be automatically conducted. The programming toolkit simplifies the application logic in the sense that it enables convenient "outsourcing" of asynchronous context processing.

Specifically, an abstract class AbstractApplication is adopted to encapsulate the interactions between the application and the middleware. An XML description for contextual properties is provided (as introduced in Section 3.2.1). A callback interface ResultCallback and a class Callback are introduced for the middleware to notify the application to automatically conduct complex context-aware behavior, as shown in Fig. 5 of the Appendix, available in the online supplemental material. The abstract class AbstractApplication can help the application register/unregister the pairs of contextual properties and context-aware behavior to MIPA (through the method register()/unregister()). The interface ResultCallback is provided for the application to conduct the context-aware behavior. The class Callback consists of a suit of ResultCallback, and is adopted for the application to register several context-aware behavior towards the same contextual property. The application can also register several contextual properties with the same context-aware behavior to MIPA. The design of callbacks provides flexibility for the development of the context-aware adaptation logic of the application.

Based on the programming toolkit, the development of context-aware adaptation logic can be simplified. The

```
public class SafetyManagementApplication extends AbstractApplication {
  public void run() {
    boolean isStarted = false;
    Naming server = MIPAResource.getNamingServer();
    RobotManager robotManager = (RobotManager) server.lookup("RobotManager");
    while (true) {
      int hour = Calendar.getInstance().get(Calendar.HOUR_OF_DAY);
      if (hour >= 8 && isStarted == false) {
        isStarted = true;
        robotManager.startPatrolling();
        Callback onAbsence = new Callback();
        ResultCallback toAlarm = new ToAlarm();
        ResultCallback toNotifyManager = new ToNotifyManager();
        onAbsence.attach(toAlarm);
        onAbsence.attach(toNotifyManager);
        register("config/predicate/absence.xml", onAbsence, "absence");
        Callback onLeak = new Callback();          register the predicate in 'absence.xml'
        ResultCallback toEscape = new ToEscape();  with the callback 'onAbsence'
        ResultCallback toCallEmergencyManager = new ToCallEmergencyManager();
        onLeak.attach(toAlarm);
        onLeak.attach(toEscape);
        onLeak.attach(toCallEmergencyManager);
        register("config/predicate/leak.xml", onLeak, "leak");
      }                                            register the predicate in 'leak.xml'
      if (hour >= 18 && isStarted == true) {       with the callback 'onLeak'
        isStarted = false;
        unregister("absence");
        unregister("leak");
        robotManager.stopPatrolling();             unregister the predicates
      }
    }
  }
}
public class ToAlarm extends UnicastRemoteObject implements ResultCallback {
  public void callback(String value) throws RemoteException {
    //sound the alarm in the workshop
  }                                                conduct context-aware behavior
}
```

Fig. 6. Code for the safety management application.

application only have to register the predicates and the corresponding callbacks in a straightforward condition-action way. Concrete applications only have to extend the class AbstractApplication, specify contextual properties in XML, and implement the corresponding context-aware behavior in the method callback() of the concrete ResultCallback. When a contextual property is detected true, the middleware will trigger the corresponding callbacks of the application automatically.

4.2 Exemplar Application

In this section, we exemplify the development of context-aware applications using the chemical plant safety management application.

Assume that the robots patrol the plant from 8:00 to 18:00. At 8:00, the application starts the robots to monitor several contextual properties for safety management. When the properties are detected true, MIPA will notify the application to conduct corresponding context-aware behavior. At 18:00, the application stops the robots. The robots are equipped with wireless modules, material leak sensors, and RFID readers (for people identification). The application is concerned with the following contextual properties:

- ϕ_1: all the robots detect no people in a workshop.
- ϕ_3: all the robots detect hazardous material leak in a workshop.

When ϕ_1 is detected true, the safety management application should conduct the following behavior:

- b_1: sound the alarm in the workshop.
- b_2: notify the plant manager of the absence.

When ϕ_3 is detected true, the safety management application should conduct the following behavior:

- b_1: sound the alarm in the workshop.
- b_3: notify the workers in the workshop to escape.
- b_4: notify the emergency managers nearby.

To property ϕ_1, we analyze its formal representation in Section 2.2.2, and show its XML description absence.xml in Fig. 5. We realize two callbacks ToAlarm and ToNotifyManager for its two context-aware behavior b_1 and b_2, respectively, as shown in Fig. 6.

To property ϕ_3, we can decompose it into three local predicates:

- $LP_1^{\phi_3}$: R_1 detects hazardous material leak.
- $LP_2^{\phi_3}$: R_2 detects hazardous material leak.
- $LP_3^{\phi_3}$: R_3 detects hazardous material leak.

Thus, the application is concerned with the snapshot predicate:

- ϕ_3: $LP_1^{\phi_3} \wedge LP_2^{\phi_3} \wedge LP_3^{\phi_3}$.

As the property ϕ_3 is a "bad thing", we adopt the modal operator $Pos(\cdot)$, as discussed in Section 2.2.2. That is, the property specified to the middleware is $Pos(\phi_3)$. Its XML description leak.xml can be found in Fig. 6 in the Appendix, available in the online supplemental material. We realize two additional callbacks ToEscape and ToCallEmergencyManager besides the callback ToAlarm for its three context-aware behavior b_1, b_3, and b_4, as shown in Fig. 6.

The code of the application is shown in Fig. 6.[6] The two callbacks corresponding to property ϕ_1 are wrapped by a callback onAbsence (an instance of the class Callback). Property ϕ_1 (in XML absence.xml) is registered to MIPA with the callback onAbsence, as shown in Fig. 6. The three callbacks corresponding to property ϕ_3 are wrapped by a callback onLeak. Property ϕ_3 (in XML leak.xml) is registered to MIPA with the callback onLeak, as shown in Fig. 6. When ϕ_1/ϕ_3 is detected true, the corresponding callbacks will be invoked automatically.

As we can see, the code of the application is reasonably clear, concise, and easy-to-understand, and the code for the registration of contextual predicates with their callbacks is quite short and simple. In comparison, a brute-force approach without the support from MIPA and the programming toolkit has to handle the tedious collection of asynchronous contexts from distributed devices and the complex reasoning over the asynchronous contexts by the application itself. The resulting code would be much more complex, thus having much less maintainability and reusability. MIPA allows the application developers to focus on the development of the context-aware behavior, while not having to explicitly cope with the distributed and asynchronous contexts. The programming toolkit provides the application developers with a convenient way to specify contextual properties and implement their corresponding context-aware behavior. The lessons learned in developing applications under the guidance of PD-CA can be found in Appendix A, available in the online supplemental material.

5 Experimental Evaluations

In this section, we evaluate the scalability of MIPA by varying the types of predicates, the number of predicates, and the number of context collecting devices. Note that

6. The code pieces for callbacks ToNotifyManager, ToEscape, and ToCallEmergencyManager are similar to ToAlarm in principle, and are omitted here.

TABLE 2
The Average Memory Consumption under Conjunctive Predicates

(a) Memory consumption of the PD-server (MB)

		The number of predicates					
		1	200	400	600	800	1000
The number of robots	20	1.7	20.9	39.3	57.0	73.9	89.9
	30	1.8	33.5	64.1	93.3	122.4	148.1
	40	1.9	51.5	97.6	141.7	185.2	225.4

(b) Memory consumption of the CC-clients (MB)

		The number of predicates					
		1	200	400	600	800	1000
The number of robots	20	2.5	23.4	43.6	62.8	81.6	99.4
	30	3.1	42.5	81.0	117.9	153.3	187.7
	40	3.4	67.8	130.1	189.7	247.5	303.7

the correctness analysis and performance evaluation of each specific predicate detection scheme with respect to different degrees of asynchrony in the computing environment have been studied in our previous work [9], [13], [17], [20]. Please refer to our previous work for detailed evaluations.

5.1 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 (3.10 GHz), and 8 GB RAM. We simulate the plant safety management scenario discussed in Section 4.2.

Our experiments evaluate the detection of three representative types of predicates: conjunctive predicate ϕ_1 (defined in Section 2.2.2), regular expression predicate ϕ_2 (defined in Section 2.2.2), and CTL predicate ϕ_4.[7] We tune the number of predicates and the number of robots, to study the memory consumptions of the PD-server and the CC-clients, and the response latency of the PD-server. We use M_s and M_c to denote the average of the memory consumption of the PD-server and of all the CC-clients through the lifetime, respectively. We use T_{rt} to denote the actual response latency of the PD-server (from the instant when the property detection process is triggered to the instant when the detection finishes) as the elapse of time, and use T_s to denote the average of the response latency of the PD-server through the lifetime.

The application registers a predicate to MIPA every second. 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 minutes, and the average interval between the activities (where the local predicate is false) is 1 minute. The context data of each robot is up to 16,000. We model the communication delay between the robots by exponential distribution with the average delay of 10 ms. The lifetime of the experiment is up to 2 hours.

5.2 Evaluation Results

In this section, we discuss the evaluation results concerning the conjunctive predicates, regular expression predicates, and CTL predicates.

7. The CTL predicate is "$\forall(a \cup b)$", indicating that at least one of the robots should be moving (i.e., the snapshot predicate 'a') until the robots are at the end point (i.e., the snapshot predicate 'b').

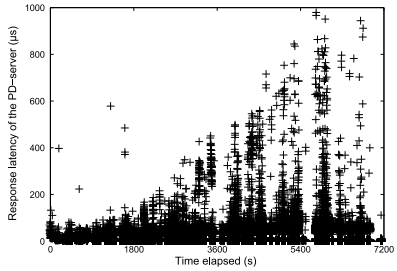


Fig. 7. The response latency as the elapse of time under conjunctive predicates.

5.2.1 Detection of Conjunctive Predicates

We first evaluate the performance of MIPA concerning the detection of conjunctive predicate ϕ_1. We tune the number of conjunctive predicates from 1 to 1,000 and the number of robots from 20 to 40.

As shown in Table 2, when fixing the number of robots, M_s and M_c increase linearly with the number of predicates. When fixing the number of predicates, M_s and M_c increase slowly with the number of robots. The slow increase of M_s and M_c is because the detection algorithm for conjunctive predicates is space-efficient [9]. When the number of robots is 40, M_s of 1,000 predicates is quite small and acceptable for a context-aware middleware. Meanwhile, M_c of 1,000 predicates is 303.7 MB. That is, the average space cost for each robot is 7.6 MB, which is acceptable on resource-constrained context collecting devices, such as mobile robots and smart phones.

We also investigate the response latency T_{rt} of MIPA as the elapse of time. We fix the number of robots to 40 and the number of predicates to 600. As shown in Fig. 7, most detections are accomplished within 100 μs, with an average of 20 μs. We also find that certain number of detection encounter abrupt increases in the latency. This is mainly because besides the cost for predicate detection, nontrivial resources are also intermittently needed to support thousands of concurrent threads dedicated to the predicate detection. We also find that the unusually long latencies of T_{st} increase linearly while remaining within 1 ms.

We further study the average response latency T_s with respect to the number of robots and the number of predicates. As shown in Fig. 8, when fixing the number of robots, T_s remains almost the same. When fixing the 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 25 μs). The reason is that the detection algorithm for conjunctive predicates is time-efficient [9]. The cost-effectiveness of the algorithm for

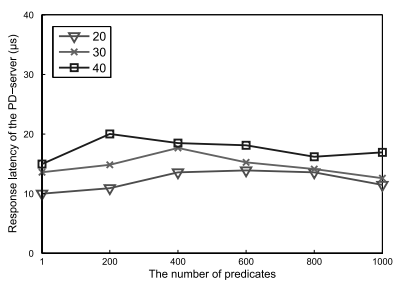


Fig. 8. The average response latency under conjunctive predicates.

TABLE 3
The Average Memory Consumption under Regular Expression Predicates

(a) Memory consumption of the PD-server (MB)

		The number of predicates					
		1	20	40	60	80	100
The number of robots	3	1.7	4.3	7.0	9.7	12.4	15.0
	4	1.9	7.9	13.0	24.1	28.0	32.2
	5	4.6	32.2	66.9	102.2	139.7	172.0

(b) Memory consumption of the CC-clients (MB)

		The number of predicates					
		1	20	40	60	80	100
The number of robots	3	2.0	2.0	2.2	2.4	2.6	2.8
	4	1.9	2.2	2.3	2.7	2.9	3.2
	5	2.0	2.1	2.4	2.7	3.0	3.4

conjunctive predicates enables MIPA to support the detection of thousands of conjunctive predicates at the same time.

5.2.2 Detection of Regular Expression Predicates

We then evaluate the performance of MIPA concerning the detection of regular expression predicate ϕ_2. We tune the number of regular expression predicates from 1 to 100 and the number of robots from 3 to 5.

As shown in Table 3a, when fixing the number of robots, M_s increases linearly with the number of predicates. When fixing the number of predicates, M_s increases quite fast with the number of robots. This is because the detection algorithm for regular expression predicates on the property detection process is exponential to the number of robots [13]. M_s of regular expression predicates is much more than that of conjunctive predicates shown in Table 2a. Thus, respecting the memory cost, the detection of regular expression predicates can only work well on a moderate scale with a limited number of context collecting devices.

As shown in Table 3b, M_c increases slowly with the number of predicates and the number of robots. When we tune the number of predicates from 1 to 100, M_c remains within 3.4 MB, and costs much less than M_s. This is because, although the detection cost for regular expression predicates on the property detection process is exponential, the algorithm on the context collecting process is linear [13]. Thus, M_c is acceptable on resource-constrained devices.

As shown in Fig. 9, we investigate the response latency T_{rt} of MIPA as the elapse of time. We fix the number of robots to 5 and the number of predicates to 60. Most detection are accomplished around 20 ms. Certain number of detection encounter abrupt increases in the latency, similar to the case of conjunctive predicates shown in Fig. 7. We also find that most detection are finished within 400 ms. The reason is that the detection algorithm for regular expression predicates can be achieved based on the *active surface* of the lattice whose size is almost stable, not having to traversing the whole lattice whose size is increasing as the elapse of time [13].

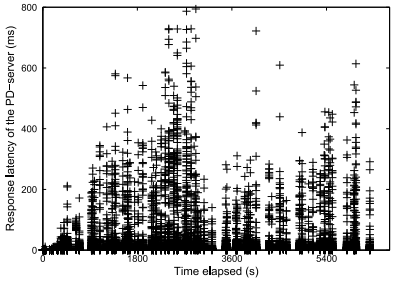


Fig. 9. The response latency as the elapse of time under regular expression predicates.

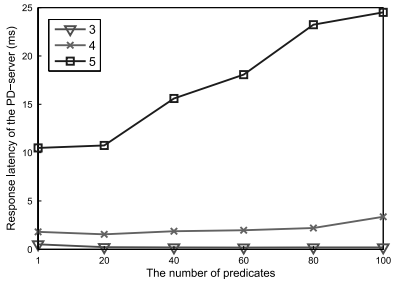


Fig. 10. The average response latency under regular expression predicates.

We also study the average response latency T_s with respect to the number of robots and the number of predicates. As shown in Fig. 10, when the number of robots is 3 or 4, T_s increases slowly and remains within 4 ms. When the number of robots is 5, T_s increases faster. We can see that T_s increases fast with the increase in the number of robots. This is in accordance with the exponential detection cost for regular expression predicates [13]. Thus, respecting the time cost, the detection of regular expression predicates also usually work on a moderate scale with a limited number of context collecting devices.

Although the detection of regular expression predicates is much more expensive than that of conjunctive predicates, we argue that in pervasive computing scenarios, the number of devices involved in one single contextual property is usually on a small scale [13]. Thus the performance of the detection of regular expression predicates is often acceptable.

5.2.3 Detection of Temporal Logic Predicates

We then evaluate the performance of MIPA concerning the detection of CTL predicate ϕ_4. We tune the number of CTL predicates from 1 to 100 and the number of robots from 3 to 5.

As shown in Table 4a, when fixing the number of robots, M_s increases almost linearly with the number of predicates. When fixing the number of predicates, M_s increases quite fast with the number of robots. The reason is that the detection algorithm for CTL predicates on the property detection process is exponential to the number of robots [17]. Compared to Table 3a, M_s of CTL predicates is much more than that of regular expression predicates. This is because the detection algorithm for CTL predicates maintains the whole lattice but the regular expression detection algorithm does not [13], [17].

TABLE 4
The Average Memory Consumption under CTL Predicates

(a) Memory consumption of the PD-server (MB)

		The number of predicates					
		1	20	40	60	80	100
The number of robots	3	1.7	4.3	7.0	9.7	12.4	15.0
	4	1.9	7.9	13.0	24.1	28.0	32.2
	5	4.6	32.2	66.9	102.2	139.7	172.0

(b) Memory consumption of the CC-clients (MB)

		The number of predicates					
		1	20	40	60	80	100
The number of robots	3	2.0	2.0	2.2	2.4	2.6	2.8
	4	1.9	2.2	2.3	2.7	2.9	3.2
	5	2.0	2.1	2.4	2.7	3.0	3.4

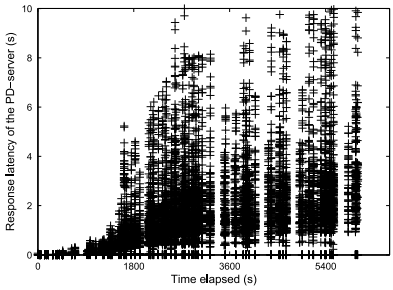


Fig. 11. The response latency as the elapse of time under CTL predicates.

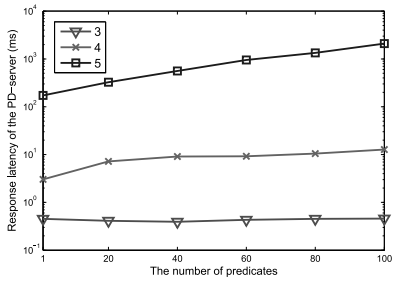


Fig. 12. The average response latency under CTL predicates.

As shown in Table 4b, M_c increases slowly with the number of predicates and the number of robots. M_c costs much less than M_s, just as that of regular expression predicates shown in Table 3b. The reason is that the detection algorithm for CTL predicates on the context collecting process is also linear [17].

We investigate the response latency T_{rt} of MIPA as the elapse of time in Fig. 11. We fix the number of robots as 5 and the number of predicates as 60. Most detection are accomplished within 4 s. The number of detection which encounters abrupt increases in the latency is more than that of regular expression predicates. This is because the detection algorithm for CTL predicates is less efficient, compared to that for regular expression predicates. Different to that of regular expression predicates in Fig. 9, the response latency T_{rt} of detecting CTL predicates as a whole increases as the elapse of time. This is because the detection of CTL predicates is achieved by traversing the whole lattice of snapshots whose size is increasing exponentially as the elapse of time [17].

We also study the average response latency T_s with respect to the number of robots and the number of predicates. As shown in Fig. 12, when the number of robots is 3 or 4, T_s remains within 10 ms. When the number of robots is 5, T_s increases quite fast from 200 ms to 2 s. We also find that T_s increases fast with the number of robots. Thus, the detection of CTL predicates can only work well on a comparatively small scale with a limited number of devices.

5.2.4 Discussions

Summarizing the evaluation results for three different types of predicates, we can also make useful observations. Specifically, when detecting different types of predicates, the memory cost on the context collecting devices is pretty low. This ensures that our system can well utilize a variety of (usually resource-constrained) context collecting devices, such as mobile robots and smart phones in pervasive computing scenarios.

On the PD-server side, the detection cost (in terms of both time and space) for snapshot properties is pretty low but the detection cost for temporal relations is much more expensive. Compared to snapshot predicates (i.e., conjunctive predicates), sequence predicates (i.e., regular expression predicates and CTL predicates) can delineate complex temporal relations of contextual events and delineate dynamic evolution of the environment. However, the expressiveness of sequence predicates are obtained by sacrificing the cost-efficiency, thus also sacrificing the range of application.

Moreover, as the universal and existential quantifications can be nested in CTL formulae, but our specification can have only one modal operator in the regular expression predicates, the detection cost for CTL predicates is much higher than that for regular expression predicates. Consequently, when specifying contextual properties, we can specify predicates concerning complex temporal relations, but the number of context collecting devices involved should be carefully decided. The number of context collecting devices which can appear in a sequence predicate is rigorously restricted by the computing capacity of the PD-server.

6 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 [21], [22], [23]. LIME enables mobile coordination by abstracting communication into the tuple space [21]. EgoSpaces enables each agent to specify its egocentric view of the network [22]. 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 [23]. Our approach, instead of maintaining available contexts for the application, enables the application to declare its concerns using hight-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 for 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". TinyDB [24] is the first project that introduced the idea of abstracting a wireless sensor network composed of TinyOS-based devices as a database. Users of TinyDB can define the desired data gathering by writing a SQL-like

query. PerLa [25] provides a SQL-like language which is able to manage heterogeneous devices and support most existing sampling modes. Payton et al. [26] 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 [27] and STEAM [28], 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.

Ontologies-based approaches provide a formal description for the entities and their relationships, and also provide support for applying ontology reasoning techniques. Chen et al. [29] propose the COBRA-ONT ontology expressed in OWL for supporting pervasive context-aware systems. COBRA-ONT is a collection of ontologies for describing places, agents, events, and their associated properties in an intelligent meeting room domain. Bhargava et al. [30] propose the RoCoMO ontology based on the Rover Context Model (RoCoM), structured around four primitives that can be used to represent and model any situation and activity: entities, events, relationships, and activities. Our work focuses on modeling the temporal order of contextual activities and reasoning contextual properties based on predicate detection.

Recently, many software methodologies for the development of context-aware applications are proposed. Dey [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 and Indulska [31] 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. [32] introduce a design methodology as well as corresponding tool support for pervasive applications. Testing and maintenance are also discussed.

In the area of predicate detection, existing work (including [7], [12], [14] and our previous work [9], [13], [17], [20]) focus on the development of efficient predicate detection algorithms for different types of predicates. In this work, we employ the predicate detection theory to achieve context-awareness in asynchronous pervasive computing environments.

7 Conclusion and Future Work

In this paper, we propose the PD-CA framework to enable context-awareness in asynchronous environments. Under the guidance of the PD-CA framework, we design and implement the MIPA middleware. We also present the programming toolkit to simplify the development of context-aware applications based on MIPA. Experimental evaluations show the performance of MIPA in enabling context-awareness despite of the asynchrony.

This work is being expanded in various directions. Fault-tolerant mechanisms, e.g., fault-tolerant predicate detection techniques, are in demand, and multiple time models, e.g., the partially synchronous model, should be integrated to cover more pervasive computing scenarios. More realistic and comprehensive experimental evaluations are also necessary.

Acknowledgments

This work is supported by the National 973 Program of China (2015CB352202), the National Science Foundation of China (61272047, 91318301, 61321491), and the program A for Outstanding PhD candidate of Nanjing University. Y. Huang is the corresponding author.

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, Apr. 2013.

[2] P. S. Duggirala, T. Johnson, A. Zimmerman, and S. Mitra, "Static and dynamic analysis of timed distributed traces," in *Proc. IEEE Real-Time Syst. Symp.*, 2012, pp. 173–182.

[3] N. D. Lane, E. Miluzzo, H. Lu, D. Peebles, T. Choudhury, and A. T. Campbell, "A survey of mobile phone sensing," *IEEE Commun. Mag.*, vol. 48, no. 9, pp. 140–150, Sep. 2010.

[4] H. Li, H. Wang, and J. Liu, "Error aware multiple vertical planes based visual localization for mobile robots in urban environments," *Sci. China Inf. Sci.*, vol. 58, no. 3, pp. 1–14, 2015.

[5] A. Dey, "Providing architectural support for building context-aware applications," Ph.D. dissertation, College of Computing, Georgia Inst. Technol., Nov. 2000.

[6] C. Cao, P. Yu, H. Hu, and J. Lv, "Toward a seamless adaptation platform for internetware," *Sci. China Inf. Sci.*, vol. 56, no. 8, pp. 1–13, 2013.

[7] A. D. Kshemkalyani and J. Cao, "Predicate detection in asynchronous pervasive environments," *IEEE Trans. Comput.*, vol. 62, no. 9, pp. 1823–1836, Sep. 2013.

[8] F. C. Gärtner, "Fundamentals of fault-tolerant distributed computing in asynchronous environments," *ACM Comput. Surv.*, vol. 31, pp. 1–26, Mar. 1999.

[9] Y. Huang, Y. Yang, J. Cao, X. Ma, X. Tao, and J. Lu, "Runtime detection of the concurrency property in asynchronous pervasive computing environments," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 4, pp. 744–750, Apr. 2012.

[10] Q. Li and D. Rus, "Global clock synchronization in sensor networks," *IEEE Trans. Comput.*, vol. 55, no. 2, pp. 214–226, Feb. 2006.

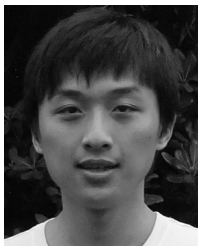
[11] R. Cooper and K. Marzullo, "Consistent detection of global predicates," in *Proc. ACM/ONR Workshop Parallel Distrib. Debugging*, 1991, pp. 167–174.

[12] V. K. Garg and B. Waldecker, "Detection of strong unstable predicates in distributed programs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 12, pp. 1323–1333, Dec. 1996.

[13] Y. Yang, Y. Huang, J. Cao, X. Ma, and J. Lu, "Formal specification and runtime detection of dynamic properties in asynchronous pervasive computing environments," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 8, pp. 1546–1555, Aug. 2013.

[14] A. Sen and V. K. Garg, "Formal verification of simulation traces using computation slicing," *IEEE Trans. Comput.*, vol. 56, no. 4, pp. 511–527, Apr. 2007.

[15] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.
[16] F. Mattern, "Virtual time and global states of distributed systems," in *Proc. Int. Workshop Parallel Distrib. Algorithms*, pp. 215–226, 1989.
[17] H. Wei, Y. Huang, J. Cao, X. Ma, and J. Lu, "Formal specification and runtime detection of temporal properties for asynchronous context," in *Proc. IEEE Int. Conf. Pervasive Comput. Commun.*, Mar. 2012, pp. 30–38.
[18] MIPA-Middleware Infrastructure for Predicate detection in Asynchronous environments. [Online]. Available: http://alg-nju.github.io/mipa/, 2015.
[19] J. Yu, Y. Huang, J. Cao, and X. Tao, "Middleware support for context-awareness in asynchronous pervasive computing environments," in *IEEE/IFIP Int. Conf. Embedded Ubiquitous Comput.*, 2010, pp. 136–143.
[20] Y. Yang, Y. Huang, J. Cao, X. Ma, and J. Lu, "Design of a sliding window over distributed and asynchronous event streams," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 10, pp. 2551–2560, Oct. 2014.
[21] 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.
[22] 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.
[23] 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.
[24] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TinyDB: An acquisitional query processing system for sensor networks," *ACM Trans. Database Syst.*, vol. 30, no. 1, pp. 122–173, Mar. 2005.
[25] F. 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, no. 2, pp. 478–496, Mar./Apr. 2012.
[26] 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.
[27] G. Chen, M. Li, and D. Kotz, "Data-centric middleware for context-aware pervasive computing," *Pervasive Mobile Comput.*, vol. 4, no. 2, pp. 216–253, 2008.
[28] 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, May/Jun. 2010.
[29] H. Chen, T. Finin, and A. Joshi, "An ontology for context-aware pervasive computing environments," *Knowl. Eng. Rev.*, vol. 18, pp. 197–207, Sep. 2003.
[30] P. Bhargava, S. Krishnamoorthy, and A. Agrawala, "An ontological context model for representing a situation and the design of an intelligent context-aware middleware," in *Proc. ACM Conf. Ubiquitous Comput.*, 2012, pp. 1016–1025.
[31] K. Henricksen and J. Indulska, "Developing context-aware pervasive computing applications: Models and approach," *Pervasive Mobile Comput.*, vol. 2, no. 1, pp. 37–64, 2006.
[32] 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, Nov./Dec. 2012.



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

Yu Huang received the BSc and PhD degrees in computer science from the University of Science and Technology of China. He is currently an associate professor in the Department of Computer Science and Technology at Nanjing University. His research interests include distributed computing theory, formal specification and verification, and pervasive context-aware computing. He is a member of the IEEE and the China Computer Federation.

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

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