Middleware Support for Context-awareness in Asynchronous Pervasive Computing Environments

Jianping Yu1,2, Yu Huang1,2+, Jianmong Cao3, Xianping Tao1,2
1State Key Laboratory for Novel Software Technology
Nanjing University, Nanjing 210093, China
2Department of Computer Science and Technology
Nanjing University, Nanjing 210093, China
csjpyu@gmail.com, {yuhuang, txp}@nju.edu.cn
3Internet and Mobile Computing Lab, Department of Computing
Hong Kong Polytechnic University, Hong Kong, China
csjcao@comp.polyu.edu.hk

Abstract—Context-awareness is an essential feature of pervasive applications, and runtime detection of contextual properties is one of the primary approaches to enabling context-awareness. However, existing context-aware middleware does not provide sufficient support for detection of contextual properties in asynchronous environments. We argue that in asynchronous environments, the concept of time needs to be reexamined. Instead of assuming the availability of global time or synchronous interaction, we should rely on logical time. To this end, we present the Middleware Infrastructure for Predicate detection in Asynchronous environments (MIPA), which supports context-awareness based on logical time. Design and operation of MIPA are explained in detail. We also evaluate MIPA with a comprehensive case study. The evaluation results show the cost-effectiveness and scalability of MIPA.

I. INTRODUCTION

Pervasive applications are typically context-aware, using various kinds of contexts, such as location and time, to provide smart services [1]. Context-aware applications need to monitor whether contexts bear specified property, thus being able to adapt to the computing environment accordingly [2]. This brings the primary issue of contextual property detection [3], [4], [5], [6]. Though detection of contextual properties has been widely studied in pervasive computing and software engineering communities, it still remains a challenging issue, mainly due to the following two observations.

Contextual properties of concern to the context-aware applications bear great variety and dynamism. Specifically, users are not only interested in local contextual properties, which can be easily obtained by singular context collecting devices, but also interested in global ones, which involve multiple decentralized devices for context collection.

Meanwhile, users are not only interested in static properties of contexts. Though static properties capture interesting aspects of contexts, they inherently lack the delineation of temporal and relative order [7], [6]. In many cases, users are also interested in behavioral properties, delineating temporal evolution of the environment.

In contrast to the variety and dynamism of contextual properties, the detection is greatly complicated by the intrinsic asynchrony in pervasive computing environments [2], [8], [5], [6], [9]. Specifically, context collecting devices do not necessarily have a global clock. They heavily rely on wireless communications, which suffer from finite but arbitrary delay. Moreover, due to resource constraints, context collecting devices (usually resource-constrained sensors) often schedule the dissemination of context data. The different context update rates also result in asynchrony.

In order to achieve context-awareness in asynchronous environments, the concept of time needs to be reexamed. Instead of assuming the availability of global time or synchronous interaction, we should rely on logical time. The basic rationale behind is to utilize the happen-before relation resulting from message causality [10], and its “on the fly” coding given by logical vector clocks [11], [12]. However, existing context-aware middleware does not provide sufficient support for detection of contextual properties in asynchronous environments.

To this end, we develop the Middleware Infrastructure for Predicate detection in Asynchronous environments (MIPA) [13]. MIPA is the first open-source context-aware middleware, which provides systematic support for coping with the asynchrony while achieving context-awareness in pervasive computing environments, as far as we know. Based on MIPA, users can flexibly specify contextual properties by different types of predicates defined over asynchronous pervasive computing environments. MIPA accepts such predicates and supports context-awareness by online predicate detection.

A comprehensive case study is conducted to evaluate MIPA. In the case study, we implement over MIPA a smart lock scenario. The evaluation results show the cost-effectiveness and scalability of MIPA in pervasive computing scenarios.

The rest of this paper is organized as follows. We first discuss runtime detection of contextual properties in asynchronous environments in Section II. Section III and Section IV present the design and operation of MIPA. Section V
presents the case study and Section VI reviews the existing work. In Section VII, we conclude the paper with a brief summary and the future work.

II. Runtime Detection of Contextual Properties in Asynchronous Environments

In this section, we first discuss how we model the asynchronous pervasive computing environment. Then we discuss how users specify contextual properties on MIPA.

Detection of contextual properties assumes the availability of an underlying context-aware middleware. The middleware accepts contextual properties specified by the application, detects at runtime whether the properties are satisfied, and informs the application of the results (see detailed discussions in Section III). Specifically, in order to detect contextual properties, a collection of non-checker processes \(P_1, P_2, \ldots, P_n\) are deployed to monitor specific regions/aspects of the environment. Examples of non-checker processes are sensor agents manipulating physical sensors. One checker process \(P_{\text{che}}\) collects context information from non-checker processes, and achieves runtime detection of contextual properties. \(P_{\text{che}}\) is usually a third-party service deployed on the context-aware middleware.

A. System Model

We model the non-checker processes as a loosely-coupled message passing system, without any global clock or shared memory. Communications suffer from finite but arbitrary delay. Dissemination of context data may also be delayed due to resource constraints. The detection of contextual properties is based on the classical Lamport’s definition of the happen-before (denoted by \(\prec\)) relation resulting from message causality [10], and its “on the fly” coding given by Mattern and Fidge’s logical vector clocks [11], [12].

Messages passed in the system can be classified into two types: i) Control message: non-checker processes send control messages among each other to establish the happen-before relation among contextual activities. This enables further detection of contextual properties confronting the asynchrony; ii) Checking message: non-checker processes send vector clock timestamps of contextual activities via checking messages to the checker process.

The execution of \(P_k\) is represented by a series of local states connected by contextual events: \(\langle s_{k0}, e_{k0}, s_{k1}, e_{k1}, s_{k2}, e_{k2}, \ldots \rangle\).\(^1\)

A global state is defined as a vector of local states with one local state from each \(P_k\). If the constituent local states are pairwise concurrent, the global state is a Consistent Global State (CGS), which denotes a meaningful observation of the global system state [14].

We define the “precede” (denoted by \(\prec\)) relation between two CGSs: \(g \prec g'\) iff \(g'\) is obtained by advancing \(g\) by exactly one step on one non-checker process. The “lead-to” relation (denoted by \(\leadsto\)) between two CGSs is defined as the transitive closure of \(\prec\). The set of all CGSs with the ‘\(\leadsto\)’ relation define a lattice[7], [14]. The run of a context-aware system can be viewed as the advancement from one CGS to the next over the lattice. We define a sequence of CGSs, where adjacent CGSs has the ‘\(\leadsto\)’ relation, as a Global Sequence (GSE).

B. Specification of Contextual Properties

The key notion of our system model is the hierarchy consisting of Local States, CGSs and GSEs [15]. Users can specify predicates on each level of the hierarchy. Specifically, i) A Local predicate is specified over contexts local to some computing device; ii) A CGS predicate consists of local predicates combined by logic operators, one from each \(P_i (1 \leq i \leq n)\) in the CGS; iii) A GSE predicate is defined as the conjunction of CGS predicates defined on the constituent CGSs.

Note that the happen-before relation we obtain is usually a partial order among the contextual events. Due to the uncertainty resulting from this partial order, we need to interpret predicates under modality \(\Phi_{\text{pos}}(\phi)\) and \(\Phi_{\text{dec}}(\phi)\). More detailed discussions on the definition of predicates can be found in [16], [7], [15].

In our model, users can use local predicates to specify properties limited to specific region the computing environment. CGS predicates can be used to specify properties of specific snapshot of the asynchronous environment. GSE predicates can be used to specify behavioral properties concerning temporal evolution of the environment. In our previous studies [5] and [6], we studied the detection algorithms for CGS and GSE predicates respectively.

III. Mapping Characteristics of Property Detection to Middleware Strategies

In this section, we first discuss the characteristics of contextual property detection in asynchronous environments. Then we discuss how such characteristics motivate design of the Middleware Infrastructure for Predicate detection in Asynchronous environments (MIPA) [13]. In the next section, we further explain the operation of MIPA.
A. Characteristics of Predicate Detection

The detection of contextual properties is transformed to the detection of logic predicates specified over the contexts. Predicate detection in asynchronous environments has the following salient characteristics:

C1. Dynamic composition of predicate checkers. Predicate detection can be viewed in a top-down manner based on the hierarchy of predicates discussed in Section II.B. In the highest level, detection of a GSE predicate requires detection of the constituent CGS predicates. It also requires that all the CGSs involved form a GSE. Similarly, detection of a CGS predicate requires detection of the constituent local predicates. It also requires all the local states involved form a CGS. Due to the hierarchical structure of predicates, the checker of an upper level predicate can be constructed from checkers for lower level predicates. Moreover, checkers for the constituent predicates may lie on multiple distributed devices. The upper level checker may need to coordinate multiple distributed checkers.

C2. Monitoring the dynamic environment. Checking a local predicate is simple in the sense that it does not require interaction among distributed devices. However, the critical issue is that checking of local predicates must capture dynamic changes in the computing environment.

C3. Integration of new sensors. Users’ requirements on context-awareness is open-ended. When new types of context collecting devices are available, users need to integrate such new devices, obtain new types of contexts, and specify predicates over such new contexts.

B. Overview of MIPA Design

Design of MIPA is motivated by the characteristics of predicate detection. From MIPA’s point of view, a pervasive computing environment is composed of an application layer, a property detection layer and a context source layer, as shown in Fig. 2. The context source layer persistently collects contexts of concern. The property detection layer receives contexts from multiple decentralized and asynchronous context sources, and detects specified contextual properties at runtime. The key components of MIPA are outlined below. They are discussed in detail in the following Section III.C to Section III.E.

Predicate detection in groups. Motivated by C1 discussed above, i) MIPA supports composition of lower level predicate checkers to obtain higher level checkers; ii) MIPA supports distributed deployment of the checkers. This is achieved by grouping of different checkers. Specifically, the grouping can be decided by the predicate structure, i.e., lower level checkers consisting the same upper level checker belong to the same group. We can also further group the checkers according to the network condition.

Contextual event notification. In order to achieve accurate and timely monitoring of the environment as required in C2, we adopt the Event-Condition-Action (ECA) mechanism. Changes in the environment are modeled as contextual “events”. Local predicates are interpreted as the “condition” to filter raw contextual events. Non-checker process serves as the event listener (“action”).

Pluggable sensor agents. To ease the inevitable process of integrating new types of context collecting devices, as required in C3, MIPA treats the sensor agents (in charge of manipulating the hardware sensors) as plug-ins. New types of sensors encode their configurations in XML, which follows the DTD specified by MIPA.

The operation of MIPA also requires a number of system components, as discussed in Section III.F.

C. Property Detection Manager

Contextual properties of concern to the application is intercepted by the property detection manager. The property detection manager employs the predicate parser to parse the properties encoded in XML and obtains structure of the predicate. Further initiation of the checkers and the contextual event notification modules depends on parsing of the predicate.

Based on the predicate structure, the manager initiates predicate checkers in a bottom-up manner. The manager contacts the context modeling and context retrieval modules to obtain where to initiate the contextual event notification modules (discussed in Section III.D). The contextual event notification module persistently updates values of the local predicates. The CGS predicate checker receives online updates of local predicate values and checks the CGS predicate. In the highest GSE level, the GSE checker collects CGS predicate values from the CGS checkers and decides whether the user-specified contextual property holds.

Grouping of the checkers is first decided by the predicate structure. Contextual event notification modules belonging to the same CGS predicate are organized in the same property detection group. Similarly, since MIPA may support concurrent checking of multiple GSE predicates, CGS checkers belonging to the same GSE checker are also organized in the same group. Grouping of the checkers can be further decided by the network condition. As we know, a pervasive application may span multiple smart spaces [1]. Communications within one smart space is usually much less expensive than those across multiple smart spaces. In light of this, if predicate checkers in the same group span different smart spaces, they can be further divided into multiple groups to reduce the expensive cross-space communications.
D. Contextual Event Notification

To cope with dynamic changes in the pervasive computing environment, we adopt the ECA mechanism [3] to achieve persistent monitoring of the environment. We model updates of the environment as contextual events. The non-checker process serves as the event listener (“action”). The non-checker process first registers the contextual events. The event condition module adopts the local predicate as the condition. The raw events are filtered by local predicates and then sent to the non-checker process. The non-checker process proactively builds the happen-before relation among contextual events, which enables further predicate detection. Then it reports local predicate values to the checkers.

E. Pluggable Sensor Agents

With the development of sensing technologies, users inevitably need to integrate new types of context collection devices, in order to obtain new types of contexts. Then users can specify predicates over the newly obtained contexts. MIPA adopts sensor agents to manipulate the hardware sensors. Sensor agents are implemented as plug-ins. When new types of sensors are available, their configurations are specified in XML, following the definition (DTD) specified by MIPA. We also need to update the context modeling module to associate raw data from the newly integrated sensor agent with existing / newly-added contextual events. The DTD of sensor configurations is listed in Fig. 3. An exemplary configuration of an RFID reader is listed in Fig. 4.

F. System Components

Operation of MIPA also relies on multiple system components, which are presented below:
Resource management. The resource manager maintains information about context collecting devices in different smart spaces. It also supports the adding and removal of context collecting devices.

Naming. When non-checker processes are initiated, the naming service facilitates the binding between the sensor agents and the corresponding context-collecting devices.

Context modeling and retrieval. The context modeling module builds the mapping between local predicates and contextual events. The context retrieval module provides information for the ECA modules to be launched for the corresponding local predicates.

IV. MIPA in Action

Based on the design of MIPA discussed above, we have developed a set of middleware components that collectively address the smart lock scenario first explored in our previous studies [17], [6]. We use this scenario (Fig. 5) to exemplify the operation of MIPA.

A. The Smart Lock Scenario

In this scenario, the smart lock application automatically locks the office when the user leaves. As discussed in Section II, we can derive the contextual property for the smart lock application by traversing the predicate hierarchy in a bottom-up manner:

Local predicate. User’s location can be easily detected by one singular type of sensor. In this scenario, we can use the RFID reader or the light sensor to detect the location. Delineation of whether the user is in the office can be easily achieved by local predicate “φ1 P = the user is detected by the RFID reader in the office”, or “φ2 R P = the user is detected by the light sensor in the office”.

CGS predicate. Obviously, the smart lock application requires accurate detection of the user’s location. However, the context is usually quite noisy [3], [4]. To improve quality of the location context, we use both the RFID reader and the light sensor to decide user’s location. To delineate that the user is in the office, we can specify the CGS predicate “φCGS = (the user is detected by the RFID reader in the office) \ (the user is detected by the light sensor in the office)”. Similarly, we can
detect whether the user is outside the office (in the corridor) by the CGS predicate \( \phi_{CGS} = (\text{the user is detected by the RFID reader in the corridor}) \land (\text{the user is detected by the light sensor in the corridor}) \).

**GSE predicate.** Though the CGS predicate can improve quality of the location context, it can only capture static status of the user (e.g., in the office). In order to delineate user’s behavior of leaving the office, we interpret this behavior as “the users is staying in the office and then he is outside the office”. We specify the GSE predicate \( \phi_{GSE} = \phi_{CGS} \sim \phi_{CGS} \) to delineate this behavior. The smart lock application is expected to lock the door when “\( \text{Def}(\phi_{GSE}) \)” becomes true.

**B. Enabling the Smart Lock Scenario with MIPA**

To enable automatic locking of the office, the user launches the smart lock application on his smart phone or laptop. The smart lock application encodes the contextual property “\( \text{Def}(\phi_{GSE}) \)” in XML and transmits it to MIPA via asynchronous message passing.

The predicate parser parses this property and obtains the predicate structure. The property detection manager finds out how to relate the constituent local predicates to contextual events from the context modeling module. For example in this scenario, the property detection manager obtains from the context modeling module that \( \phi_{1,LP} \) should be associated with contextual events from the RFID reader. Then the manager finds out where to launch the contextual event notification module associated with \( \phi_{1,LP} \) from the context retrieval module. It associates \( \phi_{1,LP} \) to the RFID reader inside the office.

When the contextual event notification module is launched, the ECA manager parses the configuration files of all sensors involved in this application, and then launches the corresponding sensor agents. The ECA manager launches the non-checker processes and starts persistent monitoring of the dynamic environment. When the non-checker process finds changes in local predicate values, it reports the results to the predicate checkers. In this scenario, the RFID reader inside the office periodically detects whether there are RFID tags within its coverage. Thus the reader generates periodical contextual events of the user is or is not in the office. The events that the user is not detected are pruned and only the events that the user is in the office are passed to the non-checker process associated with \( \phi_{1,LP} \).

The property detection manager initiates the CGS checkers. The CGS checkers wait for local predicate values from the non-checker processes. The property detection manager initiates the GSE checker, which waits for CGS predicate values from the constituent CGS checkers. In this scenario, the GSE checker for \( \text{Def}(\phi_{GSE}) \) waits for the results from the CGS checkers for \( \phi_{1,CGS} \) and \( \phi_{2,CGS} \). The checker of \( \phi_{1,CGS} \) waits for results from contextual event notification modules of \( \phi_{1,LP} \) and \( \phi_{2,LP} \). Finally, when the GSE checker detects that contextual property \( \text{Def}(\phi_{GSE}) \) holds, MIPA notifies the smart lock application. Then, the application triggers locking of the door.

**V. Case Study**

In our case study, we implement MIPA with Java SE 1.6. In our case study, we run MIPA over Sun JVM 1.6.0_16 on Gentoo Linux 10.0 (kernel 2.6.31). We use a machine with an Intel Core Duo T2450 (2.0GHz) and 2GB RAM. We implement the smart lock scenario discussed in Section IV. We first study the memory consumption and response latency of MIPA when it detects one single predicate. Then, we evaluate the scalability of MIPA. We increase the number of predicates, as well as the number of non-checker processes. We generate the user’s stay in and out of the office with the exponential distribution. The average stay in and out of the office are 10 and 5 minutes respectively. The lifetime of the case study is up to 9 hours.

**A. Memory Consumption and Latency of Detection**

In this experiment, we first study the memory consumption on both the predicate detection side (PD-side) and the contextual event notification side (ECA-side). Then, we evaluate the latency of each component on the PD-side. We evaluate both \( \phi_{CGS} \) and \( \phi_{GSE} \).

As for the memory consumption, since \( \phi_{GSE} \) consists of two \( \phi_{CGS} \), the memory consumption of \( \phi_{GSE} \) is higher than that of \( \phi_{CGS} \), as shown in Table I. The memory consumption of \( \phi_{GSE} \) is more than that of \( \phi_{CGS} \) by up to 16% on the PD-side. On the ECA-side, the memory consumption of \( \phi_{CGS} \) is approximately the same with that of \( \phi_{GSE} \). This is because though the checking of \( \phi_{CGS} \) and \( \phi_{GSE} \) is different, their contextual notification modules are quite similar.

As for the latency, we measure the latency induced by the predicate parser and the property detection manager on the PD-side, as shown in Table II. The property detection manager of \( \phi_{GSE} \) is more complicated than that of \( \phi_{CGS} \). It needs to create two CGS checkers and look up more context modeling and retrieval information than the manager of \( \phi_{CGS} \). As shown in Table II, the latency of property detection manager of \( \phi_{GSE} \) is more than that of \( \phi_{CGS} \) by up to 9.04%. Since both predicates are quite simple, the parsing time is quite similar. The total latency of \( \phi_{GSE} \) is more than that of \( \phi_{CGS} \) by 9.03%.
### TABLE I
MEMORY CONSUMPTION

<table>
<thead>
<tr>
<th></th>
<th>(\phi_{CGS}) (Kb)</th>
<th>(\phi_{GSE}) (Kb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PD</td>
<td>630</td>
<td>750</td>
</tr>
<tr>
<td>ECA</td>
<td>1209</td>
<td>1208</td>
</tr>
</tbody>
</table>

### TABLE II
LATENCY

<table>
<thead>
<tr>
<th></th>
<th>(\phi_{CGS}) (ms)</th>
<th>(\phi_{GSE}) (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicate parser</td>
<td>0.149</td>
<td>0.158</td>
</tr>
<tr>
<td>Property detection manager</td>
<td>106.701</td>
<td>117.299</td>
</tr>
<tr>
<td>Total</td>
<td>106.852</td>
<td>117.458</td>
</tr>
</tbody>
</table>

### B. Scalability

An important goal of MIPA is to ensure that the predicate detection and contextual notification modules can scale to a sufficiently large number of interactions. In this study, both the predicate detection and the contextual event notification modules run on a typical desktop machine. To put MIPA under stress, multiple predicates checkers and multiple non-checker processes are concurrently running.

1) **Scalability on the Predicate Detection Side:** In this section, we study the scalability of MIPA, in terms of memory consumption and latency, on the PD-side. As for the memory consumption, we first investigate the raw memory consumption statistics. We increase the number of predicates to 3000 at a constant speed within 500 seconds. First, we find that the memory consumption increases linearly, when the number of predicates increases linearly, as shown in Fig. 6. Then the memory consumption remains relatively stable, as the maximum number of predicates has been reached.

We also study how the average of memory consumption increases as the number of predicates increases, as shown in Fig. 7. The memory consumptions of both \(\phi_{CGS}\) and \(\phi_{GSE}\) increase linearly when the number of predicates increases. This is in accordance with the result in Fig. 6. The structure of \(\phi_{GSE}\) is more complex than that of \(\phi_{CGS}\). So the memory consumption of \(\phi_{GSE}\) is more than that of \(\phi_{CGS}\) by up to 35%.

As for the latency performance, we first increase the number of predicates to 8000 and study the changes in latency. As shown in Fig. 8, most queries are served within around 600 ms (as the number of predicates increases to 8000, the latency is 611.3 ms). The increase in the latency of most queries is linear. We also see that certain number of queries encounter abrupt increases in the latency. This is mainly because when these queries are being served, there also come predicate detection quests and nontrivial computing resources are occupied. We find that the unusually long latencies also increase linearly.

We then study the average the latency imposed by \(\phi_{CGS}\) and \(\phi_{GSE}\), as shown in Fig. 9 and 10 respectively. Since \(\phi_{GSE}\) is more complex, MIPA supports a much larger number of \(\phi_{CGS}\) when achieving similar latency performance. Overall, the latency increases almost linearly as the number of predicates increases.

2) **Scalability on the Contextual Event Notification Side:** In this section, we study the scalability of MIPA, in terms of memory consumption and latency, on the ECA-side. As for the memory consumption, we first study the raw memory consumption statistics. We increase the number of non-checker processes to 500 at a constant speed within 500 seconds. The memory consumption increases linearly when the number of non-checker processes increases, as shown in Fig. 11. When the maximum number of non-checker processes is reached, the memory consumption remains stable. The average memory consumption also increases linearly, as shown in Fig. 12. This is similar to the increase in memory consumption on the PD-side.

As for the latency performance, we find that when load on the ECA-side is small, it can serve a request of creating a new non-checker process immediately. The latency only slightly increase as the number of non-checker processes increases, as shown in Fig. 13. Similar to the latency on the PD-side, we also see that certain number of queries encounter abrupt increases in the latency. This is mainly because such queries encounter the predicate detection which greatly consume computing resources. The unusually long latencies also increase linearly.

We also study how the average latency increases. We find that the average latency increases more quickly than linearly,
as shown in Fig. 14. This is mainly because when there are more updates from the ECA-side, the load on the PD-side also increases.

The readers are referred to [5], [6], for detailed discussions on whether the predicate detection schemes for \( \phi_{CGS} \) and \( \phi_{GSE} \) work correctly, confronting different types and different levels of asynchrony in pervasive computing environments.

VI. RELATED WORK

Recently, a variety of context-aware middleware have been developed, such as Gaia [1] and Cabot [3]. Based on such middleware infrastructures, a number of contextual property detection schemes have been proposed for pervasive context, such as [3], [4]. However, such schemes implicitly assume that contexts being checked belong to the same snapshot of time, thus not working in asynchronous environments. Our MIPA supports contextual property detection in asynchronous environments.

Coping with the asynchrony has been a critical issue in pervasive computing. In his pioneering work, Anind Dey pointed out that computing devices must share the same notion of time and be synchronized to the greatest extent possible. However, in some cases, just knowing the ordering of events or causality is sufficient [2]. In [8], the possible faults which can be caused by asynchronous update of context data is investigated, but it is not discussed how to cope with the asynchrony. In our previous studies [5], [6], we investigated the design and evaluation of property detection algorithms for CGS and GSE predicates. These work are implemented as property detection services over MIPA.

Development of MIPA is based on one of our ongoing research projects. A preliminary report based on version 0.3 of MIPA is presented in [15]. In this work, we significantly extend MIPA in the following aspects. In this work, we add the property detection manager, which supports combining CGS checkers and obtaining GSE checkers. We also add detailed design of pluggable sensor agents (In [15], only CGS predicates can be implemented). Moreover, we add a smart lock scenario to explain the operation of MIPA, and the scenario is implemented over MIPA using both CGS and GSE predicates. Based on these implementations, a comprehensive case study is conducted to evaluate the cost-effectiveness and scalability of MIPA, and the evaluation results are discussed in detail.

VII. CONCLUSION

In this work, we study how to provide middleware support for achieving context-awareness in asynchronous pervasive computing environments. Toward this objective, our contributions are: i) we introduce runtime detection of contextual properties based on logical time, to cope with the asynchrony in pervasive computing environments; ii) we design and implement MIPA to provide middleware support for runtime de-
tection of contextual properties in asynchronous environments; iii) a comprehensive case study is conducted to evaluate MIPA. Currently, the MIPA middleware still suffers from several limitations. In our future work, we will investigate how to design a general algorithmic skeleton, to enhance the design and implementation of various predicate detection algorithms. We also need to study how to reduce the message complexity induced by our property detection schemes. Moreover, we will deploy MIPA to multiple distributed devices and conduct more realistic experimental evaluations.

ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China (No. 60903024, 60736015, 60721002), the National 973 Program of China (2009CB320702), and the National 863 Program (2009AA01Z117).

REFERENCES