THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

A Survey of Context-Aware Pervasive Applications: From Development Support to Quality Assurance

PhD Qualifying Exam Report

Student: Yepang Liu

Supervisor: Shing-Chi Cheung

Submission Date: December 28, 2011

Abstract – Pervasive applications aim to provide unobtrusive and reliable computing services by seamlessly integrating devices into end users' everyday life. As an important branch of pervasive computing, context-aware applications continuously sense environmental changes and automatically adapt their behaviors. In recent years, such kind of computing paradigm is becoming more and more popular with the proliferation of versatile mobile devices and increasing deployment of pervasive infrastructures. However, due to the distributed nature and heterogeneity of context sources, building and maintaining a context-aware application is a non-trivial task. Researchers have identified quite a few critical challenges for systematic engineering of these applications, ranging from modeling to quality assurance. In this survey, we focus on reviewing existing research works related to development support and quality assurance. For development support, we will introduce some famous middleware infrastructures, development toolkits as well as typical design strategies. For quality assurance, we will cover context quality enhancement, testing and software model checking approaches. In the final section of this survey, we will point out some promising research directions in this area.

Table of Contents

1. Introduction 2
2. Preliminaries 2
   2.1 Context Preliminaries 2
      2.1.1 Context Definition and Classification 2
      2.1.2 Context Modeling 3
      2.1.3 Context Acquisition 4
   2.2 Context-Aware Application Architecture and Modeling 5
      2.2.1 Three-Layered Architecture 5
      2.2.2 State Transition System Model 6
3. Development Support 7
   3.1 Context-Aware Middleware 7
      3.1.1 Gaia Meta-Operating System 8
      3.1.2 SOCAM: Service-Oriented Context-Aware Middleware 10
      3.1.3 CARISMA: Context-Aware Reflective Middleware System for Mobile Applications 12
      3.1.4 RCSM: Reconfigurable Context-Sensitive Middleware 13
   3.2 Development Toolkit and Methodology 14
      3.2.1 The Context Toolkit 14
      3.2.2 Model-Based Development 16
4. Quality Assurance 17
   4.1 Context Management 17
      4.1.1 Context Inconsistency Detection 18
      4.1.2 Context Inconsistency Resolution 20
   4.2 Model Checking 21
   4.3 Testing Context-Aware Pervasive Applications 22
      4.3.1 Applying Metamorphic Testing 22
      4.3.2 Extending Conventional Data Flow Testing Criteria 23
      4.3.3 Automated Context-Aware Test Generation 24
5. Possible Research Directions 26
6. Conclusion 27

1. INTRODUCTION

Pervasive computing, first introduced by Mark Weiser in 1991, depicts a vision where the most profound technologies weave themselves into the fabric of everyday life and are indistinguishable from it [Weiser 1991]. As an important branch of pervasive computing, context-aware applications continuously sense environmental changes, and adapt their behaviors accordingly [Abowd et al. 1999; Dey et al. 2001]. In recent years, the proliferation of mobile devices and increasing deployment of pervasive infrastructures, context-aware applications are becoming increasingly popular [Two forty four a.m. LLC 2011; Crafty Apps 2011; Inizziativa Networks 2011].

Context-aware applications have some characteristics, which distinguish them from their conventional counterparts. Firstly, they are context-driven. Context changes are intrinsically unpredictable, and the quality of sensed contextual data is hard to control [Xu et al. 2010]. Secondly, the applications are distributed in nature. Contextual data are gathered from heterogeneous sources. The supporting infrastructure typically contains a wide range of networked devices [Román et al. 2002]. Thirdly, the self-adaptation of context-aware applications is guided by a set of rules, also known as policies [Capra et al. 2003; Sama et al. 2010a]. This resembles expert systems. These characteristics make context-aware applications attractive because the applications can automate certain tasks and provide services imperceptibly. However, they also pose huge challenges for developing and maintaining such applications [Abowd 1999; Kramer et al. 2007; Cheng et al. 2009; Kulkarni et al. 2010]. Over the past two decades, worldwide researchers have done extensive research to tackle these challenges.

In this survey, we look in depth at the challenges, and the research efforts that target at addressing them. Our survey serves as a guideline for researchers who want to conduct research to make context-aware applications more powerful and reliable.

2. PRELIMINARIES

In this section, we introduce the preliminaries about context-aware pervasive applications to lay the groundwork for this survey. Accurate definition, modeling and acquisition approaches will be presented in section 2.1. Common architecture and modeling technique of context-aware applications will be discussed in section 2.2.

2.1 Context Preliminaries

2.1.1 Context Definition and Classification

How to accurately define context to confine its scope is one of the key questions facing researchers at the very beginning of the exploration in this area [Baldauf et al. 2007]. Hull et al. [1997] defined context as the aspects of the current situation. Dey [1998] referred to context as the end user's location, orientation and emotional state etc. However, definitions by analogy, as in the former one, or by enumeration, as in the latter one, will both cause unnecessary confusions as they only captured part of the nature of context. The first formal and accurate definition was given by Abowd et al. in 1999.

> "Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves".

Based on this definition, we can further classify contexts into two major categories, i.e. external and internal [Prekop and Burnett 2003]. External contexts refer to the environment's physical characteristics that can be measured by sensors, such as illuminance and temperature. Internal contexts specify the logic characteristics of the concerned entities, such as the activity and emotional state of the end user. Internal context are normally provided by end users or interpreted from information captured by sensors. For example, CyberDesk interprets the end users' emotional status by monitoring their activities [Dey 2000]. Due to the uncertainty introduced by context interpretation, many applications [Two forty four a.m. LLC 2011; Crafty Apps 2011; Inizziativa Networks 2011; Want et al. 1992; Abowd et al. 1997; Cheverst et al. 2000; Sumi et al. 1998] prefer to use external contexts such as the location data collected from GPS sensors, or user specified internal contexts like preference profiles. Readers should be alerted that interpretation is not the only source that causes context uncertainty. Sensor's reliability, heterogeneity of context sources and many other factors can pose threats to the quality of contexts.

2.1.2 Context Modeling

During the designing phase of a context-aware application, the developers need to consider carefully how to represent context. When choosing a context model, they should take the following common factors into account [Baldauf et al. 2007].

— **Simplicity.** Software engineers prefer a simple and yet effective solution to the problem at hand. A simple modeling technique will ease both the development and maintenance of context-aware applications.

— **Flexibility.** A good context model should be flexible enough in the sense that it is not limited to some specific types of context. A flexible model thus can be adjusted to fit into most applications and ease the future extension.

— **Expressiveness.** Application designers aim to find a modeling technique, which is capable of expressing complex context and yet simple enough for implementation. Such a context model has a direct impact on the capability of the application under designing.

Existing literature contains various types of context modeling techniques [Baldauf et al. 2007]. We list some representatives and discuss their own characteristics in terms of simplicity, flexibility, and expressiveness.

— **Key-Value Pair**. Being the simplest model, key-value pairs are adopted by a wide range of projects [Schilit et al. 1994; Strang et al. 2004]. However, the expressiveness of this model is restricted. A key-value pair can neither easily encode the level of certainty nor express complex context. For example, the context describing that "Tim enters the conference room where Ken and Tom are probably having a heated discussion" cannot be easily modeled by key-value pairs. As a result, context reasoning based on key-value pairs is also limited.

— **Object-Oriented Model.** Object-oriented approaches provide good encapsulation and reusability. A context object hides the details of context acquisition, aggregation and reasoning. It offers the outside world a simple interface for accessing context [Cheverst et al. 1999]. This model is highly extensible. For instance, engineers can freely incorporate new features into an inherited context class or combine some classes to create a composite one when needed.

— **Markup Scheme Model.** Markup scheme based models commonly use a hierarchy of class instances and their properties to represent contexts. Contexts modeled in

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
         xmlns:owl="http://www.w3.org/2002/07/owl#"
         xmls: example="http://cse.ust.hk/context/example#"
>
…
<example: Person rdf:ID="Tim">
      <rdfs:locatedIn rdf:resource="#conference_room">
</example:Person>
…
</rdf:RDF>
```

Fig. 1. A partial ontology example

this form can be easily understood by human beings and quickly parsed by computers. In particular, user specified contexts, known as user preference profiles, are supported by standard specifications such as Composite Capability/Preference Profiles (CC/PP) [W3C 2004a].

— **Ontology Based Model.** Compared with markup schemes, ontology provides a higher level of abstraction [Gu et al. 2005; Xu et al. 2004;]. It provides a vocabulary for representing knowledge and describing situations in a specific domain. Most importantly, context reasoning becomes easy. We use an example adapted from SOCAM [Gu et al. 2005] to express a simple context that "Tim is in the conference room". Fig. 1 presents a partial ontology written in Web Ontology Language (OWL) [W3C 2004b]. Developer can encode rules to facilitate context reasoning. For instance, if there are more than two people in a conference room where the noise level is high, then they is probably a meeting going on. Reasoning may introduce uncertainty. Therefore, a confidence level is often attached to some high level contexts, and applications should be able to handle the uncertainty.

— **Logic Based Model.** Logic based models have the highest level of formality among all techniques discussed in this section. McCarthy et al. [1997] first proposed logic based modeling technique, where contexts are represented as expressions or facts. A formal logic system is applied to manipulate contexts by adding, deleting and updating facts. Logic systems internally support inference as well as being highly expressive. Although modern context-aware applications [Two forty four a.m. LLC 2011; Crafty Apps 2011; Inizziativa Networks 2011; Helsinki 2005] seldom directly adopt a logic based context model, similar formalisms always form the theoretical foundation for context reasoning or self-adaptation (see Section 2.2.1).

2.1.3 Context Acquisition

Context-aware applications continuously sense environmental changes. Acquiring contexts in an efficient and reliable way is the first step toward high application quality. We discuss the merits and demerits of some representative context acquisition techniques below [Baldauf et al. 2007].

— **Direct Sensor Access.** Applications can directly retrieve raw data from physical or virtual sensors. Virtual sensors normally refer to some web services, such as online calendar or weather forecast [Gu et al. 2005]. With this acquisition technique adopted, the application needs to handle all communications details. The software components will be tightly coupled and therefore difficult to maintain.

— **Middleware Infrastructure.** Middlewares are widely adopted in the modern context-aware applications [Kjær 2007]. They hide the sensing details and provide a

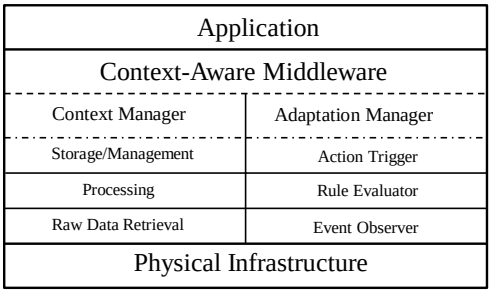


Fig. 2. Common three-layered design

simple interface for the upper layer application. Some middleware also takes care of context quality enhancement [Xu et al. 2004], ontology-based reasoning [Gu et al. 2005], context policy conflict resolution [Capra et al. 2003] and so on. Particularly, Salber et al. [1999] proposed the concept of context widgets, which mediate between the environment and the application. A widget provides contexts in a certain format to the application, and can be replaced by another one which offers the same service without affecting the application layer. For example, a location widget based on GPS can be changed to another one based on cellular networks.

In recent years, the advance in mobile technology catalyzed the emergence of handheld devices that are equipped with a variety of sensors, and powerful operating systems. Building context-aware applications for these terminals becomes easier. Developers simply need to call some APIs to obtain the contexts in need. Therefore, they can focus more on the business logic.

— **Context Server.** Context server is a centralized approach for managing context data. A context server retrieves raw data from sensors and performs some necessary computation. Applications contact the server to obtain the context in need in a synchronous or asynchronous way (see Section 2.2.1). The technique can greatly reduce network overhead, and facilitates the mobile terminals that have limited computational resources. However, with this technique adopted, backups should be available as the "single point failure" poses huge challenge to the whole system.

— **Networked Service.** Networked service is the distributed counterpart of the context server. The service components may distribute over the entire system network [Gu et al. 2005]. A service component advertises its services through broadcasting or other similar ways. Service discovery techniques are applied for other components in the system to locate the needed service. This technique is more reliable than context server at the expense of higher network overhead.

2.2 Context-Aware Application Architecture and Modeling

2.2.1 Three-Layered Architecture

Context-aware applications differ from their conventional counterparts in the need to handle context data retrieval, management and self-adaptations [Cheng et al. 2009; Korpipaa et al. 2003; Dey 2000]. In order to reduce development and maintenance costs, a multi-layered design is widely adopted as it is based on the "separation of concerns" principle [Sama et al. 2010b]. Although different applications have their own uniqueness in architecture, a three-layered design is commonly adopted.

As presented in Fig. 2, there are mainly three layers in a common design, namely physical infrastructure layer, context-aware middleware layer and the application

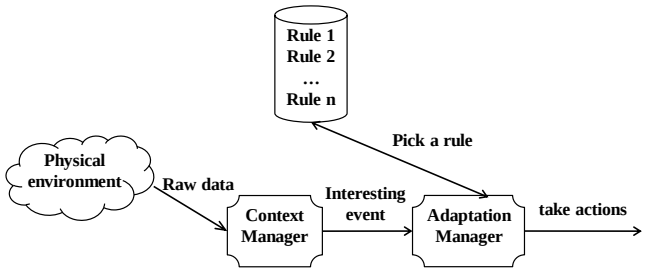


Fig. 3. Event-condition-action computing paradigm

layer. Application layer deals with the business logic. Physical infrastructure layer contains networked sensors and other distributed devices. Middleware consists of two important components.

— **Context Manager.** Context manager retrieves raw data from the physical infrastructure in a synchronous or asynchronous way. Synchronous means the context manager periodically requests context data from the physical infrastructure. This works well when the application only needs information from a small number of sensors, but does not scale to large systems due to high network and computation overhead. Context manager working in an asynchronous style subscribes to the physical infrastructure for interesting context data. The context manager will receive a message when such context data occurs. Of course, this style requires the physical infrastructure to offer subscription service.

After obtaining raw context data, the context manager will perform necessary processing to ease the computation in upper layers. For example, it may translate a Wi-Fi access point's (AP) IP address to the AP's identifier. A context manager also performs other management tasks according to an application's requirements. In some applications, valuable historical contexts should be stored and managed for a long time.

— **Adaptation Manager.** Context-aware applications adjust their operations based on context changes. Developers or end users have freedom to control the self-adaptation by explicitly or implicitly configuring rules. A rule in context-aware applications resembles those used in expert systems. Typically, a rule in "if-then" form contains two parts. Triggering condition is specified in "if" clause. Action is specified in "then" clause. Normally the condition is expressed in disjunctive normal form, and each rule has only one action. The semantics of a rule is that if current context satisfies the triggering condition, then the action should be carried out. Based on this consensus, we can see that an adaption manager should at least contain three modules: event observer, condition evaluator, and action trigger.

As a conclusion, Fig. 3 presents a systematic view of such event-condition-action (ECA) computing paradigm. The context manager retrieves and processes raw context data. After observing interesting context changes, the adaptation manager evaluates the conditions of active rules, and triggers the action of the satisfied rule.

2.2.2 State Transition System Model

The ECA computing paradigm makes it suitable to model a context-aware application as a state transition system, which is also called an Adaptation Finite-State Machine (A-FSM) in literature [Sama et al. 2010a]. Self-adaptations imply transitions

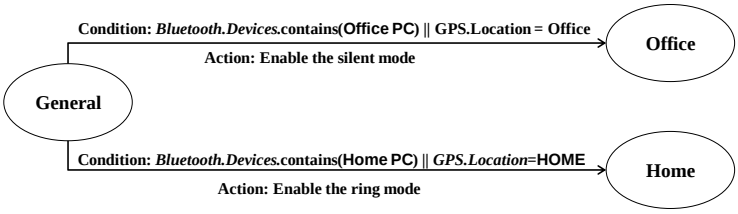


Fig. 4. State transition system example

from the current state to the new state. As an application only has a limited number of rules in practice, the number of stable states in the transition system is finite.

A state transition system is a tuple $<S, C, A, T>$. S is a finite set of stable states. C is a finite set of conditions. A is a finite set of actions. Transition $T \subseteq S \times C \times S \times A$ is a quaternary relation. If $p, q \in S$, $c \in C$ and $a \in A$, then $(p, c, q, a) \in T$ represents that there will be a transition from state p to state q if condition c is satisfied by an incoming event and the action a will be performed with the state transition. Let us consider a trivial but realistic example. Suppose an application in a smart phone is able to obtain current location and surrounding Bluetooth devices information from some context sources. The business logic is specified by two rules, which can be naturally translated to "if-then" form.

— **Rule 1.** Enable the silent mode when a Bluetooth device "Office_PC" is detected, or the phone's GPS is enabled and the current location is reported as "Office".

— **Rule 2.** Enable the ring mode when a Bluetooth device "Home_PC" is detected, or the phone's GPS is enabled and the current location is reported as "Home".

We can identify two stable states from the rules. The device is in silent mode in "Office" state, and ring mode in "Home" state. Fig. 4 gives a pictorial illustration of the state transition system model for the example application. Circles in the figure represent states and arrows correspond to state transitions. Lines above arrows describe conditions specified in rules, and lines below them define associated actions.

3. DEVELOPMENT SUPPORT

Developing context-aware applications has many challenges due to their distributed nature, unpredictable environmental dynamics, and the use of unconventional sensors. Extensive and intense research has been done for more than two decades to ease the development of such applications. Most of the efforts focus on two areas. Some research projects aim to provide infrastructural support by designing context-aware middlewares [Bellavista et al. 2003; Capra et al. 2003; Chan and Chuang 2003; Gu et al. 2005; Mckinley et al. 2005; MSU 2007; Román et al. 2002; Xu et al. 2004; Yau et al. 2002]. Other projects target at offering direct support by providing toolkits or design guidance [Salber et al. 1999; Biegel and Cahill 2004; Julien and Roman 2006; Zhang and Cheng 2006a]. Although their targets vary, those projects share similar methodology. In this section, we are going to discuss how they can help the development of context-aware applications.

3.1 Context-Aware Middleware

Traditional middlewares provide complete transparency of the underlying techniques and the running environment. Instead, context-aware middleware infrastructures have to balance between context-awareness and transparency. Applications should be aware of the changes in their running environment, and adjust their behaviors ac-

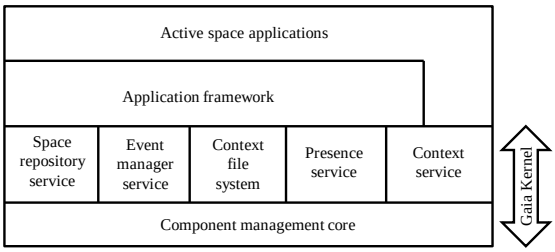


Fig. 5. Gaia architecture

cordingly. However, they need not care about the details of context sensing, communications among distributed devices in the physical infrastructure. For this reason, different middlewares offer a common set of services: (1) context acquisition, processing, (2) situation monitoring, (3) action triggering. On the other hand, different middlewares have their own uniqueness. We will introduce some representative middlewares below.

3.1.1 Gaia Meta-Operating System

Gaia is one the earliest context-aware middleware infrastructures [Román et al. 2002]. It extends traditional operating system to facilitate the construction of active spaces. An active space is a physical space coordinated by a responsive context-based software infrastructure. Users in an active space interact with their physical and digital environments seamlessly.

3.1.1.1 Gaia Architecture

Fig. 5 presents the architecture of Gaia. The Gaia kernel comprises component management core (CMC), and a set of components providing basic services for the upper layer applications. CMC dynamically manages all Gaia components and applications. Applications in active spaces are highly distributed, and require remote component execution and management. Therefore, Gaia is built on top of Corba to provide a stable infrastructure for distributed object interaction.

— **Event Manager.** In active spaces, events, such as a user's entering or a component's crash, should be monitored, and distributed to interested parties. The event manager, built upon Corba's event service, implements a decoupled communication model based on suppliers, consumers, and channels. A default set of channels notifies interested parties about basic events such as new services and component liveness. Applications can also define channels to disseminate their state changes. Decoupling event suppliers and consumers enables the event manager to offer reliable services. If some event supplier crashes, it can be replaced by a replica without affecting other components in the active space.

— **Context Service.** The context service component (CSC) supports both synchronous and asynchronous context acquisition. Interested parties in the active space query or register with CSC to obtain context data. The context infrastructure comprises a set of context providers, including sensors and higher-level context providers. Higher-level context is inferred from the context data captured by sensors. CSC uses a registry to maintain a list of available context providers. The applications are supposed to use this registry to find providers of the contexts they desire.

Context in Gaia is modeled in first order logic. Atomic contexts are expressed as first order predicates in the following form: Context(<Context Type>, <Subject>, <Relator>,<Object>). The *subject* is any entity with which the context is con-

cerned. The *object* is a property of the subject, and the *relator* associates the subject and object. For example, Context(temperature, room 4208, is, 25° C) represents that the current temperature in room 4208 is 25° C. Complex contexts can be constructed by combining atomic contexts using logic operators.

— **Presence Service.** In active spaces, presence service updates the presence information of entities. Gaia defines four major types of entities, namely applications, services, devices and people. Its digital entity presence subsystem updates the presence of application and services. Applications and services periodically send heartbeat signal to indicate their presence. If they fail to send the signal, they are assumed to be no longer available. Physical entity presence subsystem manages device and people's presence information through monitoring them. Video camera and other different sensors can be used in this subsystem.

— **Space Repository.** The space repository component (SRC) manages the resources in the active space. Resources include all softwares and hardwares in the active space, such as displays or a PDF viewer. Built upon Corba Trader, SRC relies on presence service component to learn about an entity's entering or leaving. For each resource in the active space, SRC maintains a description XML file, which contains the resource's properties. For example, a description file for a display will contain its highest resolution. When applications initiate, they are able to use constraint query language to find resources they desire, such as execution node.

— **Context File System.** End users in active space are highly mobile, and thus manually transferring personal files is troublesome. Context file system (CFS) assists users in making personal storage available in their current location. In CFS, each file is attached with context such as "situation: seminar". CFS aggregates files based on their associated context, and presents context as directories. For example, all files related to "seminar" are put in "situation: seminar" directory. In this sense, CFS constructs a virtual directory hierarchy.

  CFS is able to work in two modes: file mode and context mode. In file mode, users can browse files as in a traditional file system. In context mode, data is organized by user or application defined properties and current context. Users need not care about the physical location of their data. After a file is associated with context, it belongs to a virtual directory. Users can simply use query language to access files. For example, they can type in "/type:/pdf/situation:/ubi-seminar" to obtain all PDF files related to ubi-seminar. From this point of view, CFS is a hybrid system with both data base and file system features.

3.1.1.2 Developing Applications Upon Gaia

Active space applications receive contexts from heterogeneous sources, present their status using different devices, and automatically adapt to environmental changes. Developing such applications is challenging [Kramer et al. 2007; Cheng et al. 2009]. Gaia provides an application framework to ease this task. The framework consists of

— A distributed component-based infrastructure. The infrastructure follows the traditional Model-View-Control design, and provides functionality for manipulating application component bindings.

— A mapping mechanism for customizing applications to active spaces. Application developers need to define an application generic description (AGD), and an application customized description (ACD). AGD contains a description of application components, the minimum and maximum number of instances allowed, and com-

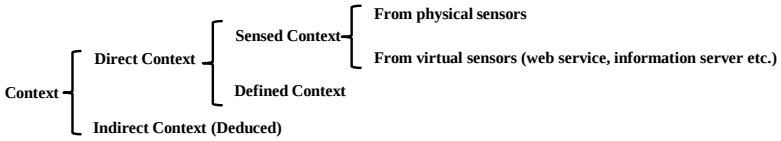


Fig. 6. SOCAM context classification

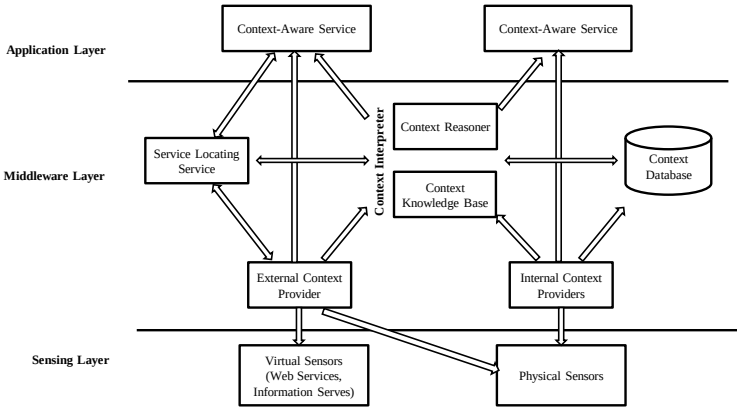


Fig. 7. SOCAM architecture

ponent requirements such as audio input. ACD describes the execution nodes and initiation parameters of application components.

— A set of policies that defines rules for customizing several aspects of applications. The application framework relies on policies to address reliability, mobility, adaptation, and related issues. Developers and users can choose to use default policies or define their own.

3.1.2 SOCAM: Service-Oriented Context-Aware Middleware

SOCAM is a distributed middleware, which provides efficient support for context discovery, acquisition, interpretation, and dissemination [Gu et al. 2005]. SOCAM uses a central context interpreter to gain context data from distributed providers, and offer it in a processed form to the clients. In SOCAM, contexts are modeled in first-order predicate calculus, and represented as domain ontology instances. Note that in Section 2.1.2, we classify SOCAM's context modeling approach as ontology based techniques. In fact, ontology has the power to express first-order predicates. So it is not contradictory that the theory foundation of context modeling is first-order logic, but contexts are represented as ontology instances. Before discussing SOCAM's architecture, we introduce its context modeling and classification as we think they are very typical.

3.1.2.1 Context Ontology Design

SOCAM's adopts a hierarchical design. There are two types of ontologies, namely generalized and domain specific ontology. Generalized ontology represents general contexts for all pervasive domains. For example, the "Device" ontology describes the properties of a general device. Domain specific ontology defines the details of general concepts and their properties for a particular domain. For example, we can inherit "Device" and define a "hands-free-phone" ontology for an automobile domain. The

separation of domains will significantly reduce the scale of context data, and ease context processing in each domain.

3.1.2.2 Context Classification

As illustrated in Fig. 6, in SOCAM, contexts are classified into two categories, namely direct context and indirect context. Indirect context is deduced from direct ones based on logic reasoning (see Section 3.1.2.3). Direct contexts include user-defined ones like user preference, and those sensed from physical or virtual sensors.

SOCAM classifies contexts in a fine granularity because different contexts have different temporal characteristics. A defined context may be valid for a long time, while a sensed context can easily be obsolete. By knowing such differences, SOCAM is able to provide better context management to enhance context quality, and perform context reasoning to maintain context consistency. Moreover, SOCAM also identifies the dependence between contexts. After knowing context dependence, SOCAM can adopt techniques such as Bayesian Network to reason about uncertain context.

3.1.2.3 SOCAM Architecture

SOCAM middleware comprises three major components: context provider, context interpreter, and service locating service. In this sub-section, we discuss the characteristics of each component, and how they communicate with each other.

— **Context Provider.** Similar to the context service component in Gaia (see Fig. 5), the context providers in SOCAM hide the low level sensing details from upper layers. Internal context providers collect contexts within a domain. Similarly, external providers collect contexts outside a domain. After processing, they provide contexts as ontology instances to the outside world.

— **Context Interpreter.** The context interpreter consists of a context reasoner and a context knowledge base. The context reasoner performs logic reasoning to deduce indirect contexts, and also maintains context consistency in knowledge base. Multiple reasoners can be incorporated to SOCAM, and each reasoned can have its own inference rules.

The context knowledge base provides a set of APIs for the applications to query, and edit contexts. Defined contexts are loaded at system initiation time, and sensed contexts are loaded at runtime. In order to guarantee the freshness of context data, SOCAM updates each piece of context periodically. Sensed contexts are updated more frequently than defined context.

The context reasoning is based on first-order logic. There are two types: ontology reasoning, and user-defined-rule-based reasoning. Ontology reasoning checks class consistency, implied relationship, and assures inter-ontology relations. SOCAM's ontology reasoning supports all the RDFS entailments described by the RDF Core Working Group, and OWL Lite. An example rule is presented below.

— Transitive Property: (?P rdf:type owl:TransitiveProperty), (?A ?P ?B), (?B ?P ?C)→(?A ?P ?C).

Ontology reasoning is important. For example, it can find out inconsistency when class A is a subclass of B, B is a subclass of C, but C is a subclass of A. The reasoning based on user-defined rules resembles ontology reasoning. The difference is that the inference rules are customized by users (see example in Section 2.1.2). Based on these rules, Socam is able to perform reasoning in different mode, including forward chaining, backward chaining, and hybrid mode.

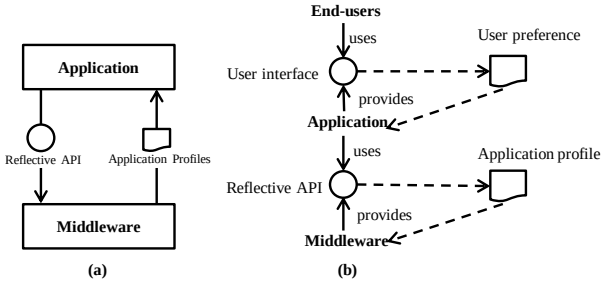


Fig. 8. CARISMA's reflective model

```
% Alice                % Bob                % Claire
MessagingService       MessagingService     messagingService
    plainMsg               plainMsg             plainMsg
        battery < 40%                           bandwidth > 50%
    encryptedMsg                            compressedMsg
        battery > 40%                           bandwidth < 50%
```

Fig. 9. Inter-profile conflict example

— **Service Locating Service.** In SOCAM, distributed context providers register with the service locating service component. The context interpreter, SOCAM's context server, uses the service locating service to locate context providers. Context providers in pervasive environment are subject to change. So the service locating service component in SOCAM has the ability to handle dynamic changes of context providers.

Finally, these distributed components communicated with each other based on Java RMI, which supports inter-operability between heterogeneous platforms, and provides a certain level of security.

3.1.2.4 Developing Applications Upon SOCAM

Developing context-aware applications on top of SOCAM is easy. The first step is to use service locating service to locate the context interpreter. SOCAM supports both synchronous and asynchronous context acquisition. The second step is to follow the ECA computing paradigm (see Section 2.1.1), and define a set of rules. The rules specify which method to invoke (action), when interesting event happens (condition). These rules are loaded into the context reasoner at system initiation time, and can be changed at runtime. One interesting thing is that one application can also register with the service locating service to provide services to other applications.

3.1.3 CARISMA: Context-Aware Reflective Middleware System for Mobile Applications

CARISMA differs from Gaia and SOCAM for its uniqueness in reflection capability and conflict resolution mechanism [Capra et al. 2003]. Each application built upon CARISMA keeps its profile as meta data in the middleware. A profile consists of passive and active part. Passive part specifies the actions, which should be performed by middleware upon the occurrence of some interesting events. Active part defines the relations between the services used by the application, and the policies that should be taken when delivering services [Kjær 2007]. In this section, we discuss how CARISMA uses reflection to adjust middleware behavior, and an microeconomic mechanism to resolve policy conflicts.

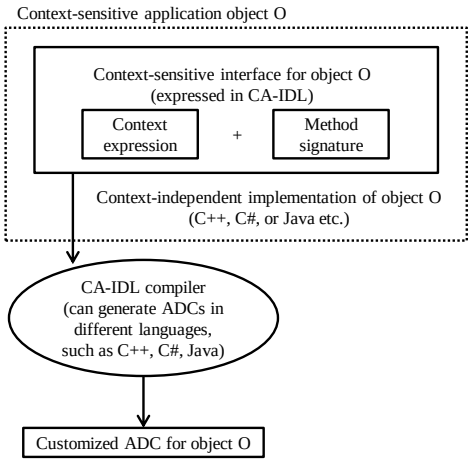


Fig. 10. ADC customization

3.1.3.1 The Reflective Model

CARISMA regards itself as a dynamically customizable service provider. It exposes some meta information, which defines its behavior, to applications. Application uses meta interface to modify the meta information in order to adjust CARISMA's behavior. As presented in Fig. 8 (a), application can modify its profile through a set of reflective APIs.

Since engineers cannot foresee all situations when designing a context-aware application for a highly dynamic environment, the end users should be granted with the permission to modify the application profile. CARISMA suggests developers to provide a friendly interface to end users. End users configure their own preference through the interface, and the application monitors such changes to modify its profile kept in the middleware accordingly. Such a layer of abstraction is necessary and very useful as end users wish to customize their application in an easy way. Fig. 8 (b) illustrates this interaction process.

3.1.3.2 The Microeconomic Mechanism

The reflective model offers flexibility for developers and end users to customize the middleware. However, it also opens door for conflicts. Consider a situation where several researchers are participating in a conference, and they would like to communicate with each other using the messaging service of their PDA, provided by the conference organizer. Suppose the messages can be delivered in three modes: plain, compressed, or encrypted. Alice and Claire have customized their application, and Bob choose the default setting. The modified profiles are given in Fig. 9. For example, Alice's PDA will send messages in plain text when the battery level is low and in encrypted form otherwise. A conflict will arise when Claire's bandwidth is lower than 50% and Alice's battery level is higher than 40%.

Interestingly, CARISMA adopts a microeconomic mechanism as its dynamic conflict resolution strategy. The auction protocol works in the following way. The middleware plays the role of the auctioneer. The applications are agents, and the good they are competing for is the execution of the policy they want most. When there is a conflict, each agent submits a sealed bid for each possible policy. The aim of the middleware is to choose a policy with the highest sum of bids received to satisfy the largest number of applications involved in the conflict.

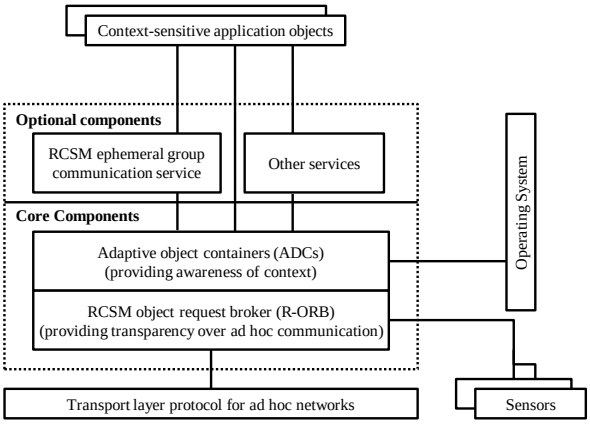


Fig. 11. RCSM architecture

3.1.4 RCSM: Reconfigurable Context-Sensitive Middleware

RCSM is designed as a middleware to facilitate the development and runtime operations of context-aware applications [Yau et al. 2002]. Like other middlewares, RCSM frees applications from context monitoring and situation detection. Developers only need to specify a context-sensitive interface, and then can concentrate on the context-independent implementation. As we have introduced many similar characteristics of context-aware middlewares, in this section we briefly discuss RCSM's uniqueness.

Fig. 11 presents the architecture of RCSM. Core components of RCSM consist of adaptive object containers (ADCs), and the RCSM object request broker (R-ORB). An ADC provides context awareness by offering runtime context monitoring, and detection services. R-ORB provides transparency over ad hoc communications among devices in the underlying network. We skip the details of R-ORB as it is not the focus of this survey.

Applications in RCSM are modeled as context-sensitive objects. Each context sensitive object O contains an interface expressed in the context-aware interface description language (CA-IDL). The interface contains a set of context expressions, which declares what kind of events are interesting, and the corresponding method signatures, which specify the actions to take upon the occurrence of interesting events. The object implementation is independent from contexts, and therefore can be developed in a conventional way.

RCSM customizes a unique ADC for each context sensitive object to provide context-awareness service. Fig. 10 sketches the ADC customization process. The CA-IDL compiler takes as input the context-sensitive interface of an object O, and outputs a customized ADC for O.

3.2 Development Toolkit and Methodology

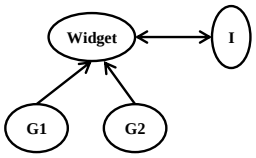
Apart from providing infrastructural support, some other research projects propose design methodologies or build toolkits to facilitate the development of context-aware application. We introduce two typical works in this section.

3.2.1 The Context Toolkit

Context Toolkit is one of the earliest research projects targeting at facilitating rapid development of context-aware applications [Salber et al. 1999]. A context widget, like

Widget Class	IdentityPresence
Attributes	
Location Identity Timestamp	Location the widget is monitoring ID of the last user sensed Time of the last arrival
Callbacks	
PersonArrives(location, identity, timestamp) PersonLeaves(location, identity, timestamp)	Triggered when a user arrives Triggered when a user leaves

(a)



(b)

Fig. 12. Context widget example

a GUI widget, is a reusable software component that provides applications with access to context information in their operating environment.

3.2.1.1 How Context Widgets Helps Developers?

Context widgets mediate between physical environment and applications. The following features of a context widget benefits software developers.

— **Transparency over context sensing details.** Like the aforementioned middlewares, a context widget hides the complexity of service discovery and context acquisition. An identity presence widget, which monitors the presence of people, can gather information using Active Badges [Want et al. 1992], video cameras, or other techniques. However, the details are transparent to applications.

— **Context abstraction.** Raw context data, such as altitude and longitude collected from GPS receiver might not be meaningful to upper layer applications. A context widget provides the service to translate them to a location name. Of course, in real world application, context abstraction goes beyond simple translation. For example, a context widget is able to perform probability-based logic reasoning to determine the on-going activity in a room.

— **Reusability and customizability.** A context widget is reusable. Once built, it can be utilized by a wide range of applications. Applications can also customize the context widgets, or combine several widgets into a composite one. For example, an activity widget might rely on a few identity presence widgets to infer the current situation in a conference room.

From the developer's point of view, a context widget class comprises a set of attributes and call back functions, as illustrated in Fig. 12 (a). Developers implement the call backs to handle interesting events.

3.2.1.2 Context Toolkit Implementation Details

Having introduced how context widgets can help rapid development of context-aware applications, we briefly discuss Context Toolkit's other characteristics.

— **Widget Composition.** Widget composition plays an important role in application development for several reasons. First, composite widgets provide richer information by aggregating contexts. Second, composing widgets helps context reasoning as we mention earlier. Third, composite widgets provides context of high quality. Comparison and consolidation of the contexts provided by different widgets helps to reduce the chance of receiving corrupted or inconsistent context.

— **Communication Mechanism.** A widget consists of multiple heterogeneous generators and interpreters as presented in Fig. 12 (b). These components need to communicate with each other. Besides, communication also exists between different widgets, or even different applications. Context Toolkit adopts a simple mechanism

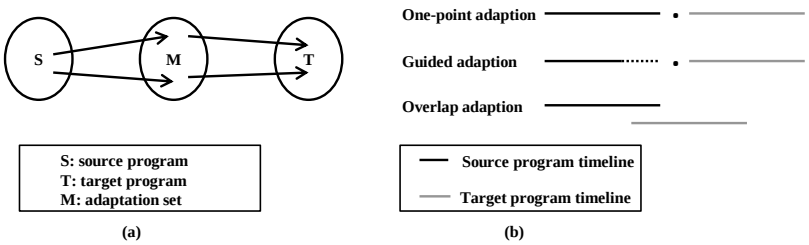


Fig. 13. Adaption styles

for communication, only assuming the underlying system supports TCP/IP and HTTP protocol. Messages are encoded in XML, which can be parsed efficiently by existing tools, such as DOM or SAX.

3.2.2 Model-Based Development

Context-aware applications are essentially self-adaptive. Developing such softwares is challenging [Kramer et al. 2007; Cheng et al. 2009]. In order to construct a reliable self-adaptive program, engineers have to guarantee that the program behavior satisfies certain expected properties before, during, and after adaptations. Model checking can be used to verify the satisfaction of properties in the expense that a formal process is adopted in the whole development process [Zhang and Cheng 2006a; 2006b]. Zhang and Cheng [2006a] proposed an approach to construct formal models for adaptive programs. Based on the general concept that a program can be presented as finite state automaton, they define an adaptive program as a program whose state space can be separated into a number of disjoint regions (each region is also viewed as a program), each of which exhibits a different steady-state behavior, and operates in a different domain. States and transitions involved in an adaptation are defined as elements of an adaptation set. As presented in Fig. 13 (a), a simple adaptation corresponds to an adaptation set. Note that, S and T in the figure does not necessarily refer to distinct programs. They could be the same piece of program with disjoint state spaces. Adaptation time point is of vital importance to the program correctness. Zhang and Cheng [2006a] proposed a concept of quiescent state, and proved that adaptation should take place in this state. The formal development process consists of six steps.

— **Step 1.** Identify the global safety and liveness properties. Global invariants are those properties which should always be satisfied, regardless of the adaptions. Normally such properties are expressed in a high-level logic language such as linear temporal logic.

— **Step 2.** Identify different domains. After each adaptation, the adaptive program enters a new domain, where it will behave differently.

— **Step 3.** Identify local safety and liveness properties. Local properties refer to the properties that should be satisfied in each individual domain.

— **Step 4.** Build a finite state automaton model for the program in each domain. Simulate the model to verify that local properties are satisfied.

— **Step 5.** Enumerate possible environmental changes, and build models for the adaption of a program from the old domain to the new one. Simulate the adaption process to verify that global invariants are satisfied, and the adaption successfully leads the program to the new domain. There are three typical adaptation styles, which help build adaptation models, as presented in Fig. 13 (b).

— **One-point Adaptation.** When the triggering condition is satisfied, the source program completes, and the target program starts. The single transition takes no time. In this style, the task for engineers is to locate the quiescent state that is suitable for adaptation.

— **Guided Adaptation.** When an adaptation is requested, the source program enters a restricted mode with limited functionalities, and keeps running until reaching a quiescent state. The key task for engineers is to identify what functionalities should be blocked in the restricted mode.

— **Overlap Adaptation.** Overlap adaption happens when the adaption consists of a sequence of transitions. This is typical when the adaptive program is multi-threaded. Each thread takes one-point or guided adaptation to finish transition. Because these transitions happen at different time, the behavior of the source and target program exists at the same time.

This formal approach is expensive. All program specifications have to be formalized. The program models must be built and modified iteratively. Visual inspection and automated model checking must be performed each time the model is changed. Although real world developers seldom adopt such a process, it still provides a great guideline to build trustworthy self-adaptive softwares.

4. QUALITY ASSURANCE

Quality assurance plays a key role in development process of a software product. Unlike conventional applications, a context-aware application incorporates the contextual information in its operating environment as part of its input. A wide range of problems emerges due to this added context-aware capability. There problems pose huge challenges to assuring the developed context-aware applications are of high quality. In this section, we introduce some representative problems, and possible solutions.

4.1 Context Management

In context-aware pervasive applications, contexts have a few characteristics, which distinguish them from the data used in conventional software [Xu and Cheung 2005].

— **Highly dynamic.** Because the application's operating environment keeps changing all the time, context may be generated in streams, and therefore easily obsolete.

— **Offered by heterogeneous sources.** The physical infrastructure of context-aware applications is distributed in nature. Contexts are often offered by a wide range of sensing units whose features such as data format and precision differ.

— **Uncertainty.** As mentioned in Section 2.1.1, context may be uncertain due to several reasons. Low reliability of sensor, context reasoning are two major sources of uncertainty.

The natural imperfectness of context leads to the common emergence of context inconsistency in real world applications [Griswold et al. 2004; Xu and Cheung 2005]. These inconsistencies reflect a contradictory understanding of the application's operating environment. For example, Dr. Green's mobile phone senses that he is now in the operating theater. A short moment later, the phone senses that a call has been missed. These two pieces of context may contradict with each other, because mobile phones are mostly not allowed to be used in operating theaters in avoidance of magnetic interference and distraction. Context quality directly affects the behavior of

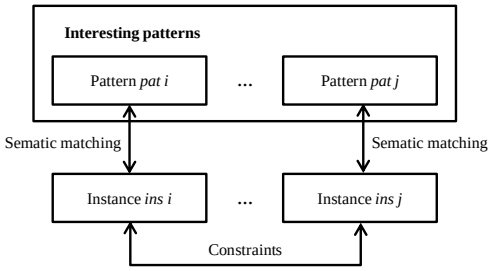


Fig. 14. Consistency checking model

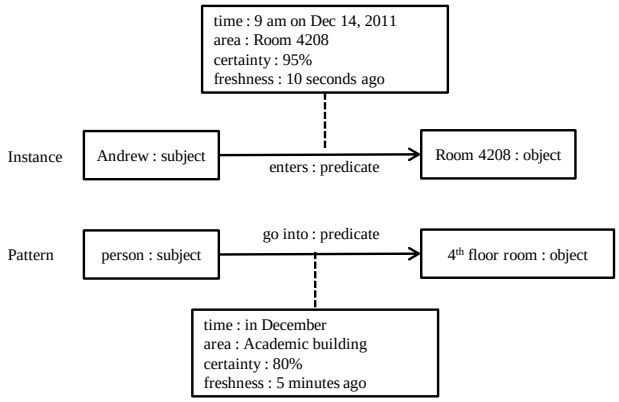


Fig. 15. Semantic matching example

context-aware applications. Therefore, inconsistencies will cause unexpected results. How to guarantee the quality of context challenges both industry and academia. In this subsection, we will introduce the state-of-the-art research efforts towards efficient and effective context management.

4.1.1 Context Inconsistency Detection

The goal of context management is to enhance context quality. The first step towards this goal is an efficient approach to detecting context inconsistencies. Context inconsistency is a sematic phenomenon instead of a syntactic one [Xu and Cheng 2005]. A set of high level constraints must be identified at prior. The constraints normally consist of a set of common sense rules, such as physical laws, and domain-specific rules provided by experts or end users. Such constraints are expressed as formal logical formulas. First-order logic, and linear temporal logic are good choices because of their high expressiveness and moderate complexity. One should realize that these constraints are only necessary conditions of consistent contexts. Satisfying them does not guarantee the perfect quality of contextual data, but violating any one of them indicates an inconsistency.

4.1.1.1 A Representative Context Consistency Checking Technique

Xu and Cheng [2005] were the first to target at context consistency checking, also known as inconsistency detection. The algorithm is based on semantic matching. In their work, context is represented as seven-field data structure *ctx = (subject, predicate, object, time, area, certainty, freshness)*. The first three fields resemble those of the first-order predicate model in Gaia (see Section 3.1.1). Time represents the effective period of the context. Area records the place with which the context is associated. Certainty is straight forward, and freshness indicates the context's generation time.

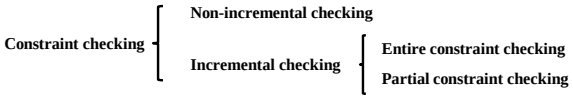


Fig. 16. Constraint checking algorithms

A context instance is defined by instantiating all field, while a context pattern is defined by instantiating some but not all fields. A semantic matching occurs if all field values in a context instance are unifiable with their counterparts in a context pattern. We use a simple example In Fig. 15 to illustrate this concept. Details about unification rules can be found in Xu et al's work [2005].

Based on semantic matching, one can design algorithms for consistency checking. Domain experts or experienced end users will pre-define some interesting application-specific context patterns, and constraints, as shown in the consistency checking model in Fig. 14. Generally speaking, consistency checking is matching a set of valid contexts with a complex context comprising a few context patterns, and checking whether any constraint is violated by the matched context instances.

4.1.1.2 Towards More Efficient Approaches

The previous section introduces a representative approach to detecting context inconsistency. Based on the proposed model in Fig. 14, many concrete checking algorithms can be derived. Unlike conventional software artifacts that tend to remain unchanged over a short period, contexts are highly dynamic and may change rapidly. Inconsistencies should be resolved with best effort before contexts are propagated into computations. Therefore, timely detection is crucial. Xu et al. [2006; 2010] further proposed more efficient algorithms. According to their taxonomy, the checking algorithms are divided into two categories, namely non-incremental, and incremental, as presented in Fig. 16. The differences between them are explained below.

— **Non-incremental checking.** When there are context changes, the whole set of constraints are re-checked to discover all detectable inconsistencies.

— **Incremental checking.** The algorithms in this category are based on the observation that not all constraints are related to context changes. For example, location contexts have nothing to do with the constraint for temperature. Therefore, only a subset of constraints needs to be rechecked. The developers need to design a way to identify the constraint subset. There are some conservative strategies proposed in literature. UML Analyzer [Egyed 2006] associates a constraint with an *instance scope* consisting of all instances of software artifacts that were accessed by the constraint. When the designer changes an UML artifact, its associated constraints will be rechecked. Of course, a new artifact may lead to the recheck of all constraints that could possibly relate to it.

  Incremental algorithms can be further divided into entire constraint checking and partial constraint checking, depending on whether the entire constraint is rechecked. Xu et al. [2010] pointed out that it is not necessary to re-check a whole constraint formula while the context changes only relate to a sub-formula. They proposed a sound algorithm to identify the part of a constraint that need to be rechecked upon context changes. Partial constraint checking is the most efficient one in the existing literature. It helps to timely detect inconsistencies in a transport network where a new context comes every 60ms in average with a miss rate of only 0.1%.

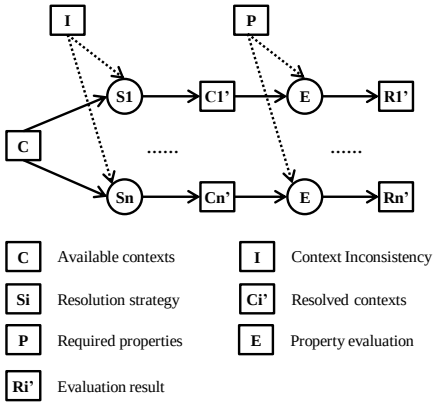


Fig. 17. Impact-oriented resolution

4.1.2 Context Inconsistency Resolution

Efficient algorithms can be used to detect most inconsistencies at runtime before contexts are fed into computation [Xu et al. 2010]. Manual resolution is impractical albeit human beings are much better at resolving semantic problem than machines. Automatic resolution techniques therefore are desirable. Similar to the *test oracle problem*, no generic approaches exist to pinpoint problematic contexts when inconsistency occurs in a set of contexts. In the literature, there are mainly the following two types of strategies.

— **Heuristics-based approach.** Domain experts or end users can define heuristic rules to resolve inconsistencies based on certain assumptions. Bu et al. [2006] suggested discarding all contexts involved in an inconsistency. Chomicki et al. [2003] proposed to discard the latest context that conflicts with existing ones. Both of the two heuristics are easy to implement, and immediately resolve inconsistencies. However, experiments based on real world contextual data show that they will cause a loss of useful contexts by 20% to 40% [Xu et al. 2008]. The application behavior would deviate much from what is expected. Xu et al. [2008] further proposed an enhanced heuristic for inconsistent resolution. Their strategy suggests that the context participating the most frequently in inconsistencies is more likely to be problematic, and should be discarded. This strategy is based on the observation that there is a time window between the generation and the usage of a context. If the contexts are always used at the generation time, the size of time window will be 0, and this strategy work the same as discarding the latest context [Chomicki et al. 2003]. This strategy is proved to be sound based on two assumptions: (1) *a set of expected contexts never cause any inconsistency;* (2) *If a set of contexts causes inconsistencies, then at least one problematic context occurs more frequently in inconsistencies than any expected context*. Experiments show that the first assumption always holds, and the second one holds in 91.7% of all cases. In addition, this strategy helps maintain 96.5% expected contexts and remove 84.7% problematic ones.

— **Analytical approach.** Context inconsistency resolution has a direct impact on an application's behavior. Different strategies have diverse adverse effects. Given a set of alternatives, it would be desirable if there is an efficient way to find out the strategy that causes the least adverse impact. Xu et al. [2007] worked towards this target, and proposed an abstract model of their on-impact oriented resolution strategy, as shown in Fig. 17. The required properties formally specify the neces-

sary conditions of correct application behavior, such as an application will get expected contexts after requesting them. On-impact oriented approach evaluates all alternatives' adverse impact on the application's behavior, and chooses the strategy with least impact to resolve inconsistency. There are two major challenges in building a tool based on this approach. Firstly, formally specified correctness properties are generally unavailable. Secondly, efficient impact evaluation algorithms, which guarantee timely inconsistency resolution, are difficult to design. For the latter, one can get useful hints from the partial constraint checking approach proposed by Xu et al. [2010]. Generally speaking, this is an interesting area worth deep exploration.

4.2 Model Checking

Context management techniques work at context level to enhance the quality of context-aware application. However, the applications are still error prone, and relies on uncertain data [Sama et al. 2010b; Cheng et al. 2009; Esfahani et al. 2011]. In this section, we explored the problems in logic level that jeopardize the correct behavior of applications, and discuss possible solutions.

The behavior of context-aware applications is driven by contexts. How the application reacts to context changes is explicitly or implicitly determined by a set of user-configured rules. The user here includes both software developers and end users. A rule typically is defined as *rule = (currentState, predicate, newState, action, priority)*. Predicate, expressed as a logic formula over a set of propositional variables, specifies the triggering condition of a rule. Example of rules can be found in Section 2.2.2 . As mentioned there, such rule-based context-aware applications can be models as state transition systems.

Configuring a set of rules without logic errors is by no means an easy job. Sama et al. [2008b; 2010a] identified five patterns of faults that commonly occur due to misconfigurations of rules and asynchronous context update. The patterns are

— **Non-determinism.** This type of faults happens when the current context changes satisfy the predicate of multiple rules with the same priority. The application is not able to determine which rule to trigger. It may randomly pick one, which may not be expected by the rule designer.

— **Dead predicate.** Rule designers may mistakenly set a rule whose predicate is internally contradictory, and therefore can never be satisfied. This tends to happen when the rules contain complex predicates.

— **Dead state.** If all the rules with the same starting state are unsatisfiable, this state itself is a dead state. When application enters this state, it will no longer have a chance to transit to other states.

— **Unstability.** Unstability occurs when the context changes cause continuous adaptations without stops, such that the application fails to stabilize in one state.

— **Unreachable state.** A state is unreachable if and only if the application can never transit to that state from the initial state.

Detecting and resolving these logic faults timely will prevent undesired consequences. Sama et al. [2010a] proposed a set of algorithms to detect faults by checking the state transition system model. The model can be derived via analyzing the rule set. In the following, we use a running example to help readers get familiar with the model checking process. The algorithm used in the example is an enumerative ver-

sion. Readers can refer to the original paper for more efficient symbolic algorithms [Sama et al. 2008a; 2010a].

In the running example, we use the rules mentioned in Section 2.2.2. The corresponding state transition system model is presented in Fig. 4. We can translate the two rules into more formal representations.

— Rule 1 = (general, $BT_{OfficePC} \vee LOC_{Office}$, office, enable silent mode, 0)

— Rule 2 = (general, $BT_{HomePC} \vee LOC_{Home}$, home, enable ring mode, 0)

The logic variable $BT_{OfficePC}$ evaluates to true if and only if the Bluetooth device "Office PC" is detected in range. Other variables are defined similarly, and we set both priorities to 0. All together, we have four logic variables ($BT_{OfficePC}$, LOC_{Office}, BT_{HomePC}, LOC_{Home}). There are 16 possible value combinations, ranging from all false's to all true's, if we ignore the dependencies among variables. An enumerative algorithm for non-determinism detection will go through each value combination. When it reaches a combination in which both $BT_{OfficePC}$ and BT_{HomePC} are true, it will conclude that if the current state is general, the situation corresponding to the value combination will lead to non-determinism.

4.3 Testing Context-Aware Pervasive Applications

Unlike conventional software whose behavior is included inside the implemented programs, context-aware applications register part of their program logic in the middleware layer. Traditional testing strategies, such as data-flow testing are not effective in revealing errors. Researchers pointed out a list of challenges, which render the traditional testing approaches not effective [Satoh 2003; Tse et al. 2004; Lu et al. 2006; Wang et al. 2007; Lu et al. 2008]. They proposed a set of novel approaches to addressing the challenges. In this section, we are going to introduce state-of-the-art research efforts.

4.3.1 Applying Metamorphic Testing

Context-aware middlewares take care of context detection, situation monitoring as well as action invocation, and therefore include part of the program logic. The multi-layered design is error prone [Sama et al. 2010b; Tse et al. 2004], and testing applications atop such middlewares has at least the following challenges.

— **Race conditions**. Middlewares and erroneous application layer program units can both change the value of some context variables. Due to environmental changes, an update from middleware may hide some computation errors, and thus make certain faults undetectable.

— **Non-testable nature of situational conditions.** Applications subscribe to middleware for interesting situations by setting conditions. Missing situations or situation relaxation can hardly be revealed by testing.

— **Unforeseeable combination of contexts**. Context changes come in an unforeseeable manner. Corresponding actions therefore can be triggered in any order. This makes the control flow of context-aware applications extremely complex.

To address the challenges, they proposed to apply *metamorphic testing* (MT). MT was originally proposed to tackle the test oracle problem [Chen et al. 1998]. Instead of relating a program output to its input, MT relates multiple input-output pairs via well-defined relations, known as *metamorphic relations*. MT suggests that even if a test case does not cause any failure, follow-up test cases can be derived using meta-

morphic relations. If any input-output pairs violate the relation, the program under test must contain errors. MT is normally used together with some test case selection strategy, which is able to generate an initial test set.

Isotropic properties of context can be used to construct metamorphic relations. For instance, similar context changes would entail similar responses from the application under test [Tse et al. 2004]. We need to consider another question: when applying metamorphic testing, who is going to identify such metamorphic relations? In reality, we can expect application designers or experienced QA team to define these relations. Therefore, the power of this technique relies on the quality of the set of metamorphic relations, and the initial test set.

4.3.2 Extending Conventional Data Flow Testing Criteria

Data flow testing is a white-box testing technique that examines the life cycle of data variables to detect improper use of data values due to implementation errors [Badlaney et al. 2006]. In data flow testing, an adequacy criterion is used to guide the test generation or selection process. Commonly used criteria are: all-def, all-uses, all-du-paths. However, traditional data flow testing is not effective in revealing errors in context-aware middleware-based applications. Lu et al. [2006; 2008] pointed out some challenges [Lu et al. 2006; 2008]. For instance, data flow in such applications will be affected by environment and context inconsistency resolution services. To make the presentation clear, let's first formally define context-aware middleware-based application. We can define such an application as a triple $<C, A, S>$, where C is a set of context variables, A is a set of adaptive actions, and S is a set of situations. For each situation $s = <C_s, p, a> \in S$, we have $a \in A$, $C_s \subseteq C$, and p is a triggering condition of s. Middleware takes care of situation detection, and invokes corresponding actions, i.e. application layer program units. Now, it is time to discuss some of the challenges.

— When traditional data-flow testing computes def-use associations, it focuses on the application layer program units, and fails to consider the definitions or uses of context variables in the middleware. Therefore, if one mistakenly sets a situation condition, traditional data-flow technique are very likely to miss the error.

— In a conventional program, the value of variables can only be changed by the program itself. However, in a context-aware application, the value context variables can also be updated by the environment. Traditional techniques fail to consider this type of variable definitions.

— Due to unforeseeable context changes and the ECA computing paradigm, the application layer program units can be invoked in a non-deterministic order. In other words, combinatorial explosion will be encountered. So it is hard to construct a control flow graph for analyzing def-use associations.

To address these challenges, Lu et al. [2006] identified two novel types of def-use associations. They are

— **Def-Situ Association.** The definition of a context variable occurs in application layer program unit, or the context value is updated by environment. The usage occurs in a situation condition, which is evaluated to true by middleware. Of course, the path from definition to usage is def-clear with respect to the context variable.

— **Pairwise Context-Aware DU Association.** This association corresponds to the circumstance where both the definition and usage of variables, not restricted to context variables, occur in application layer program units. Pairwise means we consider all

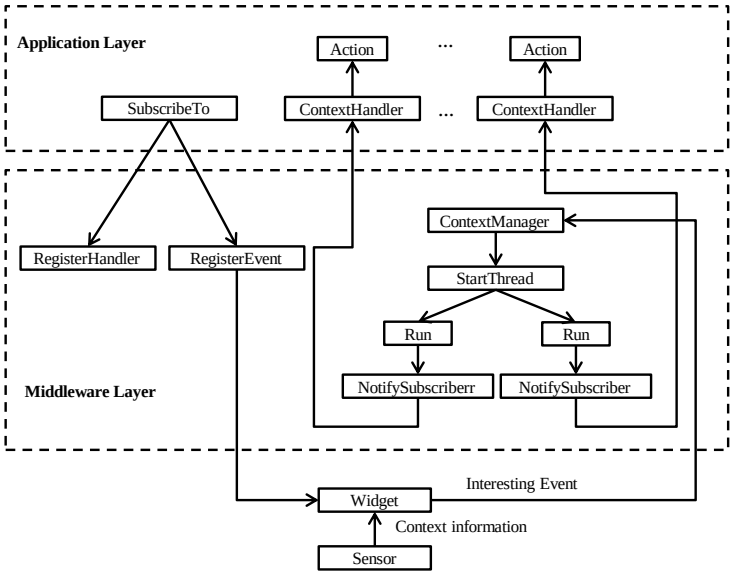


Fig. 18. A typical architecture of context-aware applications

possible combinations of two program units. Consider an action pair $<a_i, a_j>$. This association contains two subtypes. In the first type, definition occurs in a_i, and usage occurs in a_j. In the second type, both definition and usages occur in the same action.

Based on the new def-use associations, Lu et al. [2006] extend traditional data-flow testing by defining the following adequacy criteria.

— **All Situations.** This criterion requires every triggering condition to be evaluated to true at least once.

— **All Def-Situ Association.** This criterion requires the test set to cover all def-situ associations. It subsumes all situations criterion.

— **All Pairwise Context-Aware DU Associations.** This criterion requires the test set to satisfy the def-situ association criterion, and cover all pairwise context-aware du associations. It subsumes the previous two criteria.

In the most recent work, Lu et al. [2008] further proposed a set of testing adequacy criteria for testing context-aware application with inconsistency resolution services. Their observation is that context inconsistency resolution will affect data flow. For example, a discarding context service will restore the definition of a context variable to the previously killed one. Therefore, the effect on def-use associations of context inconsistency resolution services should be taken into account, and new type of associations should be defined. We do not detail this piece of work, as it adopts the same research pattern with their previous one [Lu et al. 2006].

4.3.3 Automated Context-Aware Test Generation

Incorporation of context-awareness into pervasive applications introduces a new input space, which can affect the application's behavior at any time during the application's execution. This new input space refers to the unforeseeable contextual information. It is challenging to anticipate all context changes, and when such changes can affect the application's behavior. For this reason, traditional testing approaches based on control-flow or data-flow are not effective enough to discover errors which only occur during a specific sequence of context changes. Wang et al. [2007] first pro-

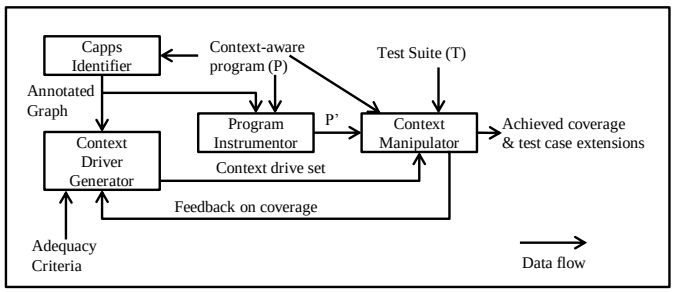


Fig. 19. Overview of automated test generation

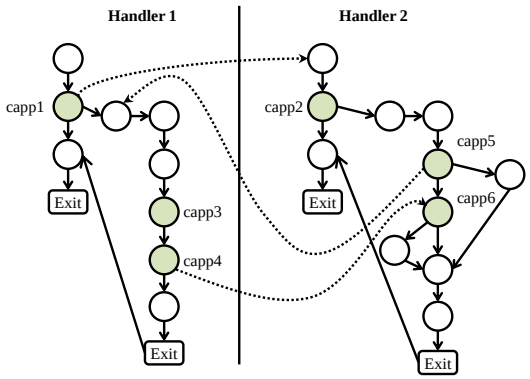


Fig. 20. Two example handlers

posed to take this issue into consideration, and presented a novel approach to improve the effectiveness of existing test suite. Their technique assumes a typical architecture of context-aware applications, as presented in Fig. 18.

In the architecture, applications implement the context handler for each type of event in which they are interested. Applications register the handlers as call backs with middleware, and register their interested events with a widget, which is a wrapper of physical sensors. When the widgets detect the occurrence of interesting events, it will notify the context manager residing in middleware. Context manager locates the corresponding handler, and starts a thread to notify the application. Then the context handler will take actions to adapt to the context change, i.e., interesting event. From this architecture, we can see that an effective testing technique should consider the streaming and unpredictable nature of contextual data, and the parallel handling of context changes.

Wang et al. [2007] proposed to use existing static analysis techniques to identify context-aware program points where context changes may affect the program behavior, and systematically manipulate the contextual data fed into the application to increase the application's exposure to context variations. Fig. 19 presents an overview of their approach.

— **Context-aware program points (capp) identifier.** This component identifies context-aware program points by analyzing the application source code. It relies on *side effect analysis* to identify statements dependent on reading or writing contextual data object fields. It also uses *escape analysis* to locate statements reading or writing objects shared among context handler threads. The identifier outputs a set

of inter-procedural control flow graphs, in which capp nodes are annotated, as shown in Fig. 20.

— **Context driver generator.** This component explores potential context handler thread interleavings, and generates a set of context drivers, which fulfills certain coverage criterion. A context driver is a sequence of capp nodes, which will be used to drive the test execution. For example, a traversal across the two handlers in Fig. 20 could generate (capp1, capp2). This driver will try to pause the execution of handler 1 at capp1, and start the execution of handler 2 in order to reach capp2. In other words, it explores one possible thread interleaving. Different coverage criteria can be defined. We introduce one example here. *StoC-k* requires the drivers to cover all possible combinations of k switches between handlers. For instance, a set D = {(capp1, capp2), (capp5, capp3), (capp3, capp3), (capp5, capp5)} satisfies *StoC-1*, because it covers all possible 1-switches between handler 1 and hander 2, i.e., {1 to 2, 2 to 1, 1 to 1, 2 to 2}. Of course, the last two switches are between two different handler threads, which happen to handle the same type of context.

— **Program instrumentor.** This component instruments the original program P by incorporating a scheduler to enable context manipulation. More specifically, it inserts a call to enterScheduler() function before each capp, and a call to exitScheduler() after each capp. The function enterScheduler() determines whether the next capp should be executed according to a context driver. If no, the current handler thread will wait for its turn to come. If yes, the capp will be executed and the exitScheduler() function will notify other waiting handler threads, and mark the capp as executed.

— **Context manipulator.** This component takes the instrumented program *P'*, a set of context drivers, and the original test suite as inputs. It runs each test case on *P'*, and tries to drive the execution towards the interesting scenarios defined by each context driver. It is possible that the manipulator fails its mission. For example, when executing the test case, none of the capps in the context driver is encountered. In this case, the achieved coverage could be lower than expected. If that happens, a new set of context drivers can be generated to guide the manipulation again in order to expose the application to more scenarios.

This automated approach can be generalized to test a wide range of context-aware applications, which adopt similar architecture to the one in Fig. 18. Of course, if test cases are not available, this approach needs to work with other testing techniques, which help generate an initial test suite.

5. POSSIBLE RESEARCH DIRECTIONS

Extensive research has been done to ease the development of context-aware applications, and enhance their quality. With the advance of mobile technologies, and the increasing deployment of infrastructures that support pervasive computing, context-awareness will be incorporated into a wide range of applications. In the future, there are many possible directions to go. We list a few of them in this section.

— Context-aware applications are commonly driven by rules. In order to help end users to configure a rule set without any logic errors, light-weight checking algorithms should be designed. We conjecture that future context-aware applications will contain a checking component, which guarantees the consistency of rules defined by end users.

— Context-aware applications commonly adopts ECA computing paradigm, and therefore resembles event-driven applications. When analyzing such applications,

enumerating all possible handler invocation sequences is impossible, given limited computational resource. Therefore, what kind of critical handler interactions should be explored remains an unsettled problem.

— Simulation is a good way to test context-aware applications. Because of the unpredictable nature of context, any context change is possible. Effective simulation should consider the model of the end user's environment. As long as the environment model is available, simulation can be done to see whether applications respond to context changes as expected. Applications developed for a specific domain, such as a museum tour side, can benefit a lot from this technique.

6. CONCLUSION

In this survey, we have studied a wide range of research projects related to context-aware pervasive applications. We introduced several typical middleware infrastructures and toolkits, which facilitate the development and runtime operations of context-aware applications. We also introduced some useful techniques that help enhance the application quality. Over the past two decades, the idea of incorporating context-awareness into applications has been widely spread. In the future, more work has to be done to further improve the reliability of such applications. In this way, context-aware applications will truly facilitate every end user's daily life by providing services imperceptibly.

REFERENCE

ABOWD, G. D. 1999. Software engineering issues for ubiquitous computing. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*. ACM, Los Angeles, CA, 75-84.

ABOWD, G. D., ATKESON, C. G., HONG, J., LONG, S., KOOPER, R., AND PINKERTON, M. 1997. Cyberguide: a mobile context-aware tour guide. *Wirel. Netw.* 3, 5, 421-433.

ABOWD, G. D., DEY, A. K., BROWN, P. J., DAVIES, N., SMITH, M., AND STEGGLES, P. 1999. Towards a better understanding of context and context-awareness. In *Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing (HUC'99)*, Springer-Verlag, London, UK, 304-307.

BALDAUF, M., DUSTDAR, S., AND ROSENBERG, F. 2007. A survey on context-aware systems. *Int. J. Ad Hoc Ubiquitous Comp.* 2, 4, 263-277.

BADLANEY, J., GHATOL, R., AND JADHWANI, R. 2006. An introduction to data-flow testing. Tech. Rep. TR-2006-22, Department of Computer Science, North Carolina State University, Raleigh, NC.

BELLAVISTA, P., CORRADI, A., MONTANARI, R., AND STEFANELLI, C. 2003. Context-aware middleware for resource management in the wireless Internet. *IEEE Trans. Soft. Engr.* 29, 12, 1086-1099.

BIEGEL, G., AND CAHILL, V. 2004. A framework for developing mobile, context-aware applications. In *Proceedings of the 2nd International Conference on Pervasive Computing and Communications (PerCom'04)*. IEEE, Washington, DC, 361-365.

BU, Y., GU, T., TAO, X., LI, J., CHEN, S., AND LV, J. 2006. Managing quality of context in pervasive computing. *In Proc. the 6th International Conference on Quality Software*, Beijing, China, 193-200.

CAPRA, L., EMMERICH, W., AND MASCOLO, C. 2003. CARISMA: context-aware reflective middleware system for mobile applications. IEEE Trans. Soft. Engr. 29, 10, 929-944.

CHAN, A. T. S., AND CHUANG, S. 2003. MobiPADS: a reflective middleware for context-aware mobile computing. *IEEE Trans. Soft. Engr.* 29, 12, 1072-1085.

CHEN, G., AND KOTZ, D. 2000. A survey of context-aware mobile computing research. Tech. Rep. TR2000-381, Department of Computer Science, Dartmouth College, Hanover, NH.

CHEN, T.Y., CHEUNG, S.C., AND YIU, S. M. 1998. Metamorphic testing: a new approach for generating next test cases, Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong.

CHENG, B. H. et al. 2009. Software engineering for self-adaptive systems: a research roadmap. *Software Engineering for Self-Adaptive Systems*. Springer-Verlag, Berlin, Heidelberg.

CHEVERST, K., DAVIES, N., MITCHELL, K., FRIDAY, A., AND EFSTRATIOU, C. 2000. Developing a context-aware electronic tourist guide: some issues and experiences. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, ACM, New York, NY, 17–24.

CHEVERST, K., MITCHELL, K., AND DAVIES, N. 1999. Design of an object model for a context sensitive tourist GUIDE. *Computers and Graphics* 23, 6, 883-891.

CHOMICKI, J., LOBO, J., AND NAQVI, S. 2003. Conflict resolution using logic programming. *IEEE Trans. on Knowledge and Data Engineering* 5, 1. 244-249.

CRAFTY APPS 2011. Tasker 2011. http://tasker.dinglisch.net/.

DEY, A.K. 1998. Context-aware computing: The CyberDesk project. In *Proceedings of the AAAI, Spring Symposium on Intelligent Environments*, Menlo Park, CA.

DEY, A.K. 2000. Providing Architectural Support for Building Context-Aware Applications. PhD thesis, Georgia Institute of Technology.

DEY, A. K. 2001. Understanding and using context. *Personal and Ubiquitous Computing* 5, 1, 4-7.

EGYED, A. 2006. Instant consistency checking for the UML. In Proceedings of the 28th International Conference on Software Engineering. 381–390.

ESFAHANI, N., KOUROSHFAR, E., AND MALEK, S. 2011. Taming uncertainty in self-adaptive software. In *Proceedings of the 19th International Symposium on the Foundations of Software Engineering (FSE'11)*. ACM, Szeged, Hungary, 234-244.

GRISWOLD, W. G., SHANAHAN, P., BROWN, S. W., BOYER, R., RATTO, M., SHAPIRO, R. B., AND TRUONG, T. M. 2004. ActiveCampus: Experiments in Community-Oriented Ubiquitous Computing. *Computer* 37, 10, 73-81.

GU, T., PUNG, H. K., AND ZHANG, D. Q. 2005. A service-oriented middleware for building context-aware services. Journal of Network and Computer Applications 28, 1, 1-18.

Helsinki 2005. ContextPhone. http://www.cs.helsinki.fi/group/context/.

HULL, R., NEAVES, P., AND BEDFORD-ROBERTS, J. 1997. Towards situated computing. In *Proceedings of the International Symposium on Wearable Computers*.

INIZZIATIVA NETWORKS 2011. Sweet Dreams 2011. https://market.android.com/details?id=com.inizz.

JULIEN, C., AND ROMAN, G. C. 2006. EgoSpaces: facilitating rapid development of context-aware mobile applications. *IEEE Trans. Softw. Engineering* 32, 5, 281-298.

KJÆR, K. E. 2007. A survey of context-aware middleware. In *Proceedings of the 25th Conference on IASTED International Multi-Conference: Software Engineering (SE'07)*. ACTA Press, Anaheim, CA, 148-155.

KORPIPAA, P., MANTYJARVI, J., KELA, J., KERANEN, H., AND Malm, E.J. 2003. Managing context information in mobile devices. *IEEE Pervasive Computing* 2, 3, 42-51.

KRAMER, J., AND MAGEE, J. 2007. Self-managed systems: an architectural challenge. In *Proceedings of Conference on Future of Software Engineering (FOSE'07)*. Minneapolis, MN, 259-268.

KULKARNI, D., AND TRIPATHI, A. 2010. A framework for programming robust context-aware applications. *IEEE Trans. Softw. Engr.* 36, 2, 184-197.

TWO FORTY FOUR A.M. LLC 2011. Locale 2011. http://www.twofortyfouram.com/.

LU, H., CHAN, W.K., AND TSE, T.H. 2006. Testing context-aware middleware-centric programs: a data flow approach and an RFID-based experimentation. In *Proceedings of the 14th International Symposium on Foundations of Software Engineering (FSE'06)*. ACM, Portland, Oregon, USA, 242-252.

LU, H., CHAN, W.K., AND TSE, T.H. 2008. Testing pervasive software in the presence of context inconsistency resolution services. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, Leipzig, Germany, 61-70.

MCCARTHY, J., AND BUVAC 1997. Formalizing context (expanded notes). In *Working Papers of the AAAI Fall Symposium on Context in Knowledge Representation and Natural Language*. American Association for Artificial Intelligence, Menlo Park, California, 99–135.

MCKINLEY, P. K., STIREWALT, R. E. K., CHENG, B. H. C., DILLON, L. K., AND KULKARNI, S. 2005. RAPIDware: component-based development of adaptive and dependable middleware. Project Rep., Department of Computer Science and Engineering, Michigan State University, East Lansing, Michigan.

MSU 2007. RAPIDware: Component-Based Development of Adaptable and Dependable Middleware. http://www.cse.msu.edu/rapidware/.

PREKOP, P., AND BURNETT, M. 2003. Activities, context and ubiquitous computing. *Special Issue on Ubiquitous Computing Computer Communications* 26, 11.

ROMÁN, M., HESS, C. K., CERQUEIRA, R., RANGANATHAN, A., CAMPBELL, R. H., AND NAHRSTEDT, K. 2002. A middleware infrastructure for active spaces. 2002. *IEEE Pervasive Computing* 1, 4, 74-83.

SALBER, D., DEY, A. K., AND ABOWD, G. D. 1999. The context toolkit: aiding the development of context-enabled applications. In *Proceedings of the SIGCHI Conference on Human factors in computing systems: the CHI is the limit (CHI '99)*. ACM, New York, NY, 434-441.

SAMA, M., ELBAUM, S., RAIMONDI, F., ROSENBLUM, D. S, AND WANG, Z. 2010a. Context-aware adaptive applications: fault patterns and their automated identification. *IEEE Trans. Softw. Engr.* 36, 5, 644-661.

SAMA, M., RAIMONDI, F., ROSENBLUM, D., AND EMMERICH, W. 2008a. Algorithms for efficient symbolic detection of faults in context-aware applications. In *Proceedings of 23th International Conference on Automated Software Engineering – Workshop*, 1-8.

SAMA, M., ROSENBLUM, D. S, WANG, Z., AND ELBAUM, S. 2008b. Model-based fault detection in context-aware adaptive applications. In *Proceedings of the 16th International Symposium on Foundations of Software Engineering (FSE'08)*. ACM, Atlanta, Georgia, USA, 261-271.

SAMA, M., ROSENBLUM, D. S, WANG, Z., AND ELBAUM, S. 2010b. Multi-layer faults in the architectures of mobile, context-aware adaptive applications. *J. Syst. Softw.* 83, 6, 906-914.

SATOH, I. 2003. A testing framework for mobile computing software. *IEEE Trans. Softw. Engr.* 29, 12, 1112- 1121.

SCHILIT, B., ADAMS, N., WANT, R. 1994. Context-aware computing applications. In *the Workshop on Mobile Computing Systems and Applications,* 85-90.

STRANG, T., AND LINNHOFF-POPIEN, C. 2004. A context modeling survey. In *First International Workshop on Advanced Context Modeling, Reasoning and Management (UbiComp 2004).*

SUMI, Y., ETANI, T., FELS, S., SIMONET, N., KOBAYASHI, K., AND MASE, K. 1998. C-map: Building a context-aware mobile assistant for exhibition tours. In *Community Computing and Support Systems, Social Interaction in Networked Communities*, Springer-Verlag, London, UK, 137–154..

TSE, T.H., AND YAU, S.S. 2004. Testing context-sensitive middleware-based software applications. In the *Proceedings of the 28th Annual International on Computer Software and Applications (COMPSAC '04).* IEEE, Washington, DC, 458-466.

W3C 2004a. Composite Capability/Preference profiles (CC/PP): Structure and Vocabularies 1.0. http://www.w3.org/TR/CCPP-struct-vocab/.

W3C 2004b. OWL Web Ontology Language. http://www.w3.org/TR/owl-features/.

WANG, Z., ELBAUM, S., AND ROSENBLUM, D. S. 2007. Automated generation of context-aware tests. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07).* IEEE, Washington, DC, USA, 406-415.

WANT, R., HOPPER, A., FALCÄO, V., AND GIBBONS, J. 1992. The active badge location system. *ACM Trans. Inf. Syst.* 10, 1, 91-102.

WEISER, M. 1991. The computer for the 21st century. *Scientific American* 265, 3, 94-104.

XU, C., AND CHEUNG, S. C. 2005. Inconsistency detection and resolution for context-aware middleware support. In *Proceedings of the 13th International Symposium on Foundations of Software Engineering (FSE'05).* ACM, Lisbon, Portugal, 336-345.

XU, C., CHEUNG, S. C., AND CHAN, W. K. 2006. Incremental consistency checking for pervasive context. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06).* ACM, Shanghai, China, 292-301.

XU, C., CHEUNG, S. C., CHAN, W. K., AND YE, C. 2007. On impact-oriented automatic resolution of pervasive context inconsistency. In *Proceedings of the 15th International Symposium on the Foundations of Software Engineering (FSE '07).* ACM, Dubrovnik, Croatia, 569-572.

XU, C., CHEUNG, S. C., CHAN, W. K., AND YE, C. 2008. Heuristics-based strategies for resolving context inconsistencies in pervasive computing applications. In *Proceedings of the 28th International Conference on Distributed Computing Systems (ICDCS'08).* IEEE, 713-721.

XU, C., CHEUNG, S. C., CHAN, W. K., AND YE, C. 2010. Partial constraint checking for context consistency in pervasive computing. ACM Trans. Softw. Engr. Methodol. 19, 3.

XU, C., CHEUNG, S. C., LO, C., LEUNG, K. C., AND WEI, J. 2004. Cabot: on the ontology for the middleware support of context-aware pervasive applications. In *Proceedings of the IFIP Workshop on Building Intelligent Sensor Networks (BISON'04).* Wuhan, China, 568-575.

YAU, S. S., KARIM, F., WANG, Y., WANG, B., AND GUPTA, S. K. S. 2002. Reconfigurable context-sensitive middleware for pervasive computing. IEEE Pervasive Computing 1, 3, 33-40.

ZHANG, J., AND CHENG, B. H. C. 2006a. Model-based development of dynamically adaptive software. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06).* ACM, Shanghai, China, 371-380.

ZHANG, J., AND CHENG, B. H. C. 2006b. Using temporal logic to specify adaptive program semantics. *J. Syst. Softw.* 79, 10, 1361-1369.