SCIENCE CHINA
Information Sciences

• REVIEW •
Special Focus on High-Confidence Software Technologies

May 2015, Vol. 58 052102:1–052102:14
doi: 10.1007/s11432-015-5300-3

A survey on dependability improvement techniques for pervasive computing systems

YANG WenHua[1,2], LIU YePang[3], XU Chang[1,2*] & CHEUNG S. C.[3*]

[1]*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing* 210023*, China;*
[2]*Department of Computer Science and Technology, Nanjing University, Nanjing* 210023*, China;*
[3]*Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong, China*

Received November 4, 2014; accepted February 6, 2015; published online March 20, 2015

Abstract The goal of this survey is to summarize the state-of-the-art research results and identify research challenges of developing and deploying dependable pervasive computing systems. We discuss the factors that affect the system dependability and the studies conducted to improve it with respect to these factors. These studies were categorized according to their similarities and differences in hope of shedding some insight into future research. There are three categories: context management, fault detection, and uncertainty handling. These three categories of work address the three most difficult problems of pervasive computing systems. First, pervasive computing systems' perceived environments, which are also called their contexts, can vary intensively, and thus have a great impact on the systems' dependability. Second, it is challenging to guarantee the correctness of the systems' internal computations integrated with interactions with external environments for developers. Fault detection is then an important issue for improving dependability for these systems. Last but not least importantly, pervasive computing systems interact with their environments frequently. These interactions can be affected by many uncertainties, which can jeopardize the systems' dependability. After a discussion of these pieces of work, we present an outlook for its future research directions.

Keywords pervasive computing system, dependability, context, fault detection, uncertainty

Citation Yang W H, Liu Y P, Xu C, et al. A survey on dependability improvement techniques for pervasive computing systems. Sci China Inf Sci, 2015, 58: 052102(14), doi: 10.1007/s11432-015-5300-3

1 Introduction

Since the proposal of pervasive computing in Weiser's seminal paper in the 1990s [1], there has been tremendous progress in the field, for example, hardware progress. As we have seen, in recent years, various handheld and wearable devices (e.g., smartphones) equipped with powerful network interfaces and rich sensors have become part of people's daily lives, providing smart and customized services. These changes result in many viable commercial and educational pervasive computing systems [2,3]. However, despite such fascinating progress, developing and deploying dependable pervasive computing systems still remains as a challenging task [4,5]. In fact, existing studies disclosed that many real-world pervasive computing systems do not provide dependable services, causing poor user experiences or economic losses [6,7].

* Corresponding author (email: changxu@nju.edu.cn, scc@cse.ust.hk)

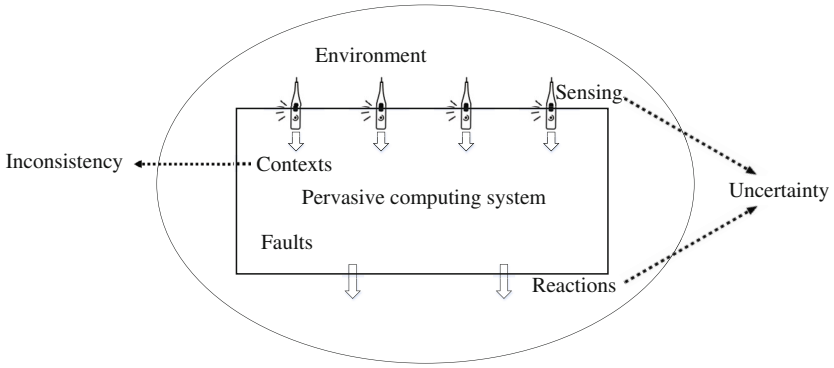

Figure 1 Overview of a pervasive computing system and its faced dependability challenges.

A pervasive computing system is context-aware, which means the system is cognizant of its environment, and able to make reaction based on this information [4]. The dependability of a pervasive computing system can be affected by multiple factors, including context inconsistency, faults in the system, and uncertainty introduced from the interaction with environment. These factors, which are depicted in Figure 1, are further discussed below.

First, pervasive computing systems intensively use contexts to capture dynamic changes from their environments and adapt their behavior accordingly. Contexts such as temperature, speed, and location are obtained from sensors. However, due to uncontrollable environmental noises, these obtained contexts may be contaminated or imprecise, which can result in context inconsistencies [6,8,9]. The inconsistencies could further affect pervasive computing systems' behavior, leading to unexpected results and reducing the dependability of such systems. Timely detection and resolution of context inconsistencies is critical. Techniques addressing these issues can be largely divided into two groups. The first one concerns how to detect context inconsistencies by checking predefined consistency constraints (or constraints for short). The second one focuses on how to find a strategy that can effectively and efficiently resolve detected inconsistencies.

Second, compared with traditional software, pervasive computing systems need to interact with complex and dynamic environments. This poses big challenges on building dependable pervasive computing systems because developers are expected to consider every detail of such interactions in a predictive way. However, this is an impractical assumption. Existing studies reported that real-world pervasive computing systems often suffer from various faults [10–12]. Unfortunately, due to their peculiarity, many of these faults cannot be effectively detected by existing quality assurance techniques for traditional softwares. For example, many pervasive computing systems perceive environment using radio frequency identification (RFID) [13–17], a key enabling technology for Internet of Things. Many faults in such RFID systems can easily bypass existing quality assurance techniques [16,17], because these faults can be either very specific to environmental interactions or are caused by special programming paradigms of pervasive computing systems. Furthermore, pervasive computing systems running in certain environments need to satisfy specific requirements. For instance, energy efficiency is a critical requirement for RFID systems and mobile applications [18].

Third, uncertainty poses stiff challenges to the dependability of pervasive computing systems and appears in various facets. Pervasive computing systems closely interact with complex and dynamic environments through devices that can probe and adapt to environmental changes (e.g., sensors and actuators). These devices can be unreliable and defective in reality. Such uncertain factors could negatively affect environmental interactions, jeopardizing the dependability of pervasive computing systems. To ease the understanding of uncertainty in the environmental interaction, we use an RFID system as an example. Typically, in such a system, there are numerous RFID tags distributed for object labeling and RFID readers deployed for tag reading. Ideally, each reader should be able to read all tags in its vicinity, but the correct reading rate of a reader may fall below 70% in real-life RFID deployments, which means at

least 30% of its contexts derived from RFID reads can be incomplete or imprecise [19]. This is one kind of uncertainty, and there also exist other kinds of uncertainties caused by unreliable sensing or flawed adaptation in pervasive computing systems [7,20].

To address the above dependability problems, researchers have expended significant effort in studying pervasive computing systems from different aspects. In this article, we systematically survey these research efforts. Specifically, we categorize existing studies into three categories with respect to the above three factors: context management, fault detection, and uncertainty handling. This criterion of categorization focuses on the essential factors affecting system dependability and covers most related research work. We summarize and compare the techniques for each category. We additionally discuss some open issues in the field, hoping to inspire future research on further improving the dependability of pervasive computing systems. The major purpose of this survey can be summed up to a three-folded contribution as listed below:

• We propose a taxonomy scheme, which can serve as a framework for easy classification of existing work on improving pervasive computing systems' dependability.

• We discuss the challenges of building dependable pervasive computing systems and provide a categorization of related state-of-the-art dependability improvement techniques. We also discuss and compare these techniques.

• We point out open issues in the field and give an outlook for future research.

The remainder of the article is organized as follows. Section 2 discusses context management techniques for pervasive computing systems. Section 3 summarizes techniques for detecting faults in pervasive computing systems. Section 4 reviews the research progress on uncertainty handling for pervasive computing systems. Section 5 highlights open issues and gives a vision of future research, and finally Section 6 concludes the article.

2 Context management

Context, a piece of information that captures the characteristics of environments, is a key concept in pervasive computing systems. It drives these systems to adapt to their dynamic running environments. To support pervasive computing systems' usage of contexts, many context modeling [21–24], management [25,26], processing, and reasoning [27,28] techniques have been proposed. For example, Schmidt et al. [21] presented a layered processing model in which sensor outputs are transformed into cues that comprise a set of values with certain measurements. Roman et al. [25] proposed to leverage a framework or middleware as a tool to manage contexts in a centralized manner to ease the use of contexts. Context Toolkit [27] aided software developers to capture context data from sensors and interpret them in an application-specific way.

In real-life systems, contexts are often obtained from heterogeneous sources in highly dynamic and noisy environments with unreliable network connections. Due to this reason, the obtained contexts can be error-prone and inconsistent with each other [6,8,9]. This prevents pervasive computing systems from correctly understanding their running environments and properly adapting to environmental changes.

Many studies have been conducted to address the context inconsistency problem [6,8,19,29]. In summary, these studies proposed two categories of techniques: context inconsistency detection techniques and context inconsistency resolution techniques. For example, Cabot [6,29] is a context management middleware for pervasive computing systems. It owns built-in consistency management services for efficient context inconsistency detection and automated inconsistency resolution. In the following, we discuss and compare these techniques in detail.

2.1 Context inconsistency detection

Detecting inconsistencies in contexts resembles detecting inconsistencies in traditional software artifacts, such as UML models [30], XML documents [31], and data structures [32]. Context inconsistency is a semantic phenomenon rather than a syntactic one, and its detection requires nontrivial reasoning

work [6,9]. Informally, we call a set of contexts inconsistent when they collectively violate a consistency constraint. Consistency constraints define necessary properties on interesting contexts and are often inferred from physical laws or user requirements. They are the key information that can help detect context inconsistencies in pervasive computing systems.

To detect context inconsistencies by constraint checking, it is necessary to properly model contexts and their consistency constraints. Existing studies [21–24] proposed many context modeling approaches, which adopt different abstraction strategies. One general approach is to model a context as a tuple that has multiple fields [9]. Each field itself can also be a tuple. These fields specify the context's type, content, and other properties and can be extended to include more information. Here, we introduce a typical context model *CM* proposed in [9]:

$$CM := \langle \textit{category, fact, restriction, timestamp} \rangle.$$

The field *category* specifies a context's type; $\textit{fact} := \langle \textit{subject, predicate, object} \rangle$ gives the content of this context, where the *predicate* associates the *subject* and the *object*; $\textit{estriction} := \langle \textit{lifespan, site} \rangle$ specifies the temporal and spatial properties of this context, in which *lifespan* specifies the validity period of the context, and *site* is the place where it occurs; and *timestamp* records the generation time of this context. A context is defined by instantiating all fields of the *CM*, and a context pattern is defined by instantiating some or all fields of the *CM*. By this definition, a context itself can also be a pattern.

On the other hand, context consistency constraints can be specified in many types of languages, such as propositional logic- and first-order logic-based languages. For example, the syntax of a first-order logic based language can be as follows:

$$\begin{aligned} f := {} & \forall v \in S\ (f) \mid \exists v \in S\ (f) \mid (f) \text{ and } (f) \mid (f) \text{ or } (f) \mid (f) \text{ implies } (f) \mid \\ & \text{not } (f) \mid \textit{bfunc}(v, \ldots, v). \end{aligned}$$

Set S represents a finite set of contexts, and the formula f is defined recursively, containing standard first-order logic operators and *bfunc* terminals. A *bfunc* terminal refers to a user-defined function that accepts contexts as input and returns a truth value. Every *bfunc* terminal is a predicate defined over one or more fields of one context or of several different contexts. For example, a *bfunc* terminal can be a function that determines whether two contexts share the same subject. After properly modeling contexts and consistency constraints, inconsistency detection is essentially a constraint checking process.

Using constraint checking to detect inconsistencies in software artifacts has been extensively studied by the software engineering community. The proposed techniques check the software artifacts against a set of predefined constraints whenever the software artifacts change. Context inconsistency detection essentially adopts the same idea, but has a critical requirement of efficiency. This is because contexts can change very frequently in real-life systems. Timely detection and resolution of context inconsistencies is a must to guarantee the dependability of such systems.

Constraint checking techniques can be classified in terms of their different detecting manners as shown in Figure 2, which is adapted from [33]. According to how existing context inconsistency detection techniques perform constraint checking, they can be divided into two categories: nonincremental checking techniques and incremental checking techniques. Whenever there is a change in a given set of contexts, nonincremental checking techniques check the contexts against every consistency constraint to find out inconsistencies. As a comparison, incremental checking techniques only check the contexts against a subset of consistency constraints, which is selected by certain strategies [8,9], upon context changes. Therefore, incremental checking techniques are typically more efficient than nonincremental checking techniques.

We can further refine the classification of incremental checking techniques. Some incremental checking techniques would recheck an entire constraint as a unit, when a consistency constraint needs rechecking [8,9]. They are called "entire constraint checking (ECC) techniques". Since contexts come in streams and change rapidly in pervasive computing systems, rechecking entire constraints may still be inefficient in some cases. To improve efficiency, other incremental checking techniques only check part of a consistency constraint when it requires rechecking if the previous checking results of the remaining part of the

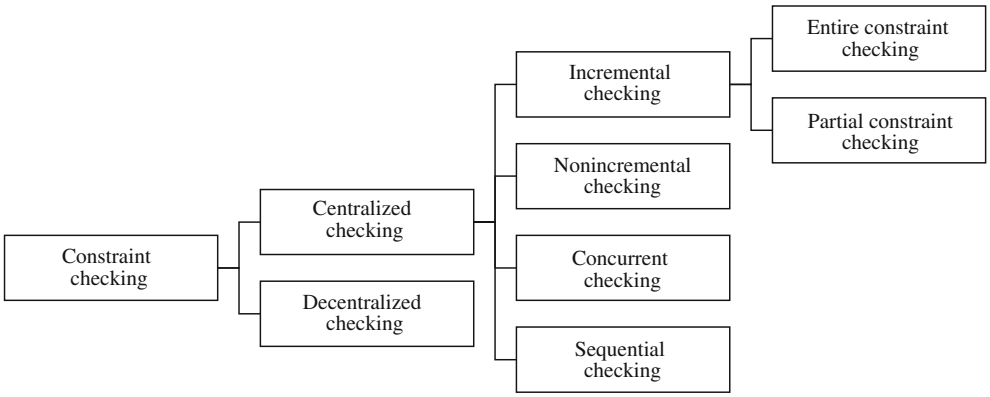

Figure 2 Classification of constraint checking techniques.

constraint are reusable. This kind of technique is called partial constraint checking (PCC) in our earlier work [9].

The basic idea of PCC is to identify the part of a consistency constraint that is irrelevant to a given context change, reuse the remaining part's previous checking results, and recheck the remaining part. The irrelevant part can be identified based on the types of context changes and a constraint's content. Let us look at a simple example. Suppose that there is a constraint $\forall x \in D\ (x > 10)$, where D is a set of contexts and x is a context in the set. The original truth value of this constraint is *true*. When a context in D is discarded (a.k.a., a kind of context change), PCC does not have to recheck this constraint. Because this formula's truth value will not change after the context discarding changes, PCC can thus reuse the constraint's previous checking result. In our earlier work [9], we also proposed algorithms to check consistency constraints and generate links to explain why certain consistency constraints are satisfied or violated. Experiments show that PCC reports more inconsistencies and less false positives compared with ECC. It is worth mentioning that since PCC heavily reuses the previous checking results of a constraint to avoid some unnecessary rechecking, it needs extra memory to maintain reusable constraint checking results (i.e., trading space cost to gain efficiency). One limitation of PCC is that it can generate many redundant links, which waste computational resources (e.g., CPU and memory). In [34], we further enhanced PCC and optimized its constraint checking by suppressing redundant links generated from inconsistency detection. This largely saves time and space cost for detecting inconsistencies and storing detection results.

The above discussed constraint checking techniques follow a centralized approach, which may cause space overhead and privacy threats to resource-limited mobile devices. Techniques introduced in [35] support constraint checking in a decentralized manner. It helps protect the privacy of distributed hosts in context inconsistency detection. To further improve the effectiveness and efficiency of context inconsistency detection, our previous work proposed a concurrent constraint checking algorithm [33]. It exploits multicore computing capability to systematically improve checking efficiency. Our recent work [36] further improved the efficiency and scalability of concurrent constraint checking by exploiting graphics processing unit computing capability. For these PCC-alike techniques, while it makes sense to improve efficiency of constraint checking, we also need to pay enough attention to the correctness of checking results. In [37], we noticed that many detected inconsistencies do not indicate real context problems. Instead, they are caused by improper inconsistency detection (e.g., improper detection timing). We can suppress the detection of such inconsistencies (named unstable inconsistencies) that do not indicate real problems by pattern learning and constraint analysis [37].

2.2 Context inconsistency resolution

Detecting context inconsistencies is the first step that allows pervasive computing systems to obtain a consistent picture of what they aim to handle. The next step is to resolve detected context inconsistencies.

Table 1 Comparison of different context inconsistency resolution strategies.

Strategy	Basic idea	Pros	Cons
Drop-latest	Discards the latest context that causes inconsistency	Simple and efficient	Insufficient, may not give desirable resolution
Drop-all	Discards all contexts relevant to any inconsistency	Simple and efficient	Discards more contexts than necessary
User-preferred	Follows user preferences to resolve inconsistencies	User-defined and flexible	Results' quality depends on user policies
Drop-bad	Discards contexts that are most likely to be faulty	Can support effective inconsistency resolution	Needs extra computing and judgment
Effect	Automatically chooses a solution to minimize impact	Flexible and effective	Expensive, needs to compare all solutions
Hybrid-fixing	Postpones the decision for inconsistency resolution	Can increase resolution accuracy	Hard to decide the postponed time for resolution

In recent years, researchers have proposed various resolution techniques and heuristic strategies to enable automatic context inconsistency resolution [38–44]. There are roughly six mainstream strategies, as shown in Table 1. Ranganathan et al. [38] and Insuk et al. [39] attempted to follow user preferences or policies to resolve inconsistencies in context evaluations, which may not be sound but can work in some cases. Bu et al. [40] proposed a drop-all resolution strategy that discards all contexts relevant to detected inconsistencies. It assumes that the contexts relevant to inconsistencies are incorrect and conservatively discards all of them for safety. Chomicki et al. [41] suggested discarding the latest context if it causes any inconsistency, which is known as the drop-latest resolution strategy. It assumes that the collection of all existing contexts is consistent, and that a new context is permitted to be included in this collection only if it does not cause any inconsistency with existing contexts. Both the assumptions may not hold in practice. This is because usually context inconsistencies are caused only by a subset of contexts (i.e., not all of them), and these contexts are not necessarily the latest ones (i.e., they may already exist before a new context is obtained). So in such cases, drop-all will discard more contexts than necessary and drop-latest may discard wrong contexts that are actually of high quality.

Another heuristic strategy drop-bad was formulated in [42], which aims to be applicable to more scenarios in pervasive computing systems. It tries to identify contexts that participate more frequently in inconsistencies. These contexts are more likely to be incorrect, and thus are considered bad contexts. Existing studies and experiments showed that in many cases, the drop-bad strategy works better than other heuristic-based techniques. Furthermore, research work [43,44] attempted to resolve context inconsistencies from a pervasive computing system's perspective. When a strategy discards some contexts to resolve inconsistencies, it may change a pervasive computing system's behavior unexpectedly because it cannot anticipate how these contexts will be used by the system. In other words, this strategy may produce side effect. In view of this, researchers proposed a strategy *effect* to resolve context inconsistencies by automatically choosing the solution that minimizes the side effect [43,44].

In practice, context inconsistency resolution may not filter out all problematic contexts. Some problematic contexts may still affect the pervasive computing systems and cause potential failures. To address this problem, Chen et al. [45] proposed a hybrid-fixing approach to resolving context inconsistencies by combining low-level context inconsistency resolution with high-level system error recovery mechanisms. It postpones the resolution to make use of future system semantics and context information to help calculate the probability that a context is problematic.

The above discussed strategies and techniques are mainly for a single pervasive computing system. Work by Yang et al. [46] further extends them for multiple pervasive computing systems that run on the same middleware. The proposed technique relies on consistent context views to isolate different pervasive computing systems, and thus makes their resolving requirements conflict-free. To ease comparison, we further summarize the basic idea, pros and cons of each strategy in Table 1.

3 Fault detection

Faults commonly exist in software artifacts. They reduce the dependability of software artifacts to a large extent. Pervasive computing systems can suffer from faults that exist in traditional software artifacts. What is worse, since pervasive computing systems continually adapt to environmental changes in an autonomic way, their adaptation may be faulty when the complexity of modeling all environmental changes is beyond a developer's capability [7,47]. For example, RFID systems, a typical kind of pervasive computing systems, are very difficult to implement because developers have to carefully consider, predict, and handle various environmental factors such as cross-coupling of neighboring RFID tags. Thus, in practice, these systems often contain many faults and are error-prone [13–17,48]. To help developers improve the dependability of pervasive computing systems, much research effort has been devoted to study possible faults in such systems and techniques for diagnosing such faults. In this section, we first discuss and categorize the faults studied in the literature, and then survey the techniques proposed for fault detection.

3.1 Faults categorization

Existing literature disclosed that many faults can occur in pervasive computing systems. Here, we roughly classify these faults into several major categories according to their behavioral characteristics. This categorization facilitates a clearer understanding of faults in pervasive systems and facilitates designing effective techniques for fault detection.

Determinism. Determinism requires a pervasive computing system's adaptation behavior to be predictable upon any situation, that is, absence of nondeterminism. For example, many pervasive computing systems are designed based on a set of adaptation rules [7,47,49], which specify the actions that the systems should execute when certain conditions are satisfied at certain system states. A system is considered to be deterministic if there is at most one rule that can be triggered at any situation. However, some pervasive computing systems' rules may be triggered simultaneously at runtime due to faults in the rule design or system implementation. This will lead to nondeterminism and cause unpredictable behavior in pervasive computing systems [10,11,50]. Many existing studies consider such nondeterminism faults harmful [11,38]. To detect these *nondeterminism faults*, Sama et al. proposed a model checking approach to verifying that all adaptations in a pervasive computing system should be deterministic [11,50]. Our previous work enhanced this model checking approach by removing false positives from its verification results through modeling the running environment of a pervasive computing system [10,51,52]. Other related studies detect nondeterminism faults at runtime and fix or tolerate them as requested [39,41,53].

Liveness and reachability. Reachability and liveness are properties commonly studied in model checking. They are also checked for pervasive computing systems' correctness. To ease understanding of the two properties, we use the above-mentioned rule-based systems for illustration again. Reachability requires each system state to be reachable from the initial state of a pervasive computing system. Liveness requires a pervasive computing system to be able to leave from any system state and trigger a corresponding rule within a finite period of time (i.e., without getting stuck in a state forever). These two properties are extensively studied for pervasive computing systems in recent years [10,11,50–52], and different checking algorithms have been proposed. Violating these properties is a strong indicator of potential faults in such systems.

Stability. As we discussed earlier, pervasive computing systems make adaptation to environmental changes smartly. Stability requires a pervasive computing system to be stable after each adaptation. Otherwise, a pervasive computing system may continually adapt itself without reaching a stable state, thus forming a long adaptation trace or even running into an infinite adaptation loop. As a consequence, the system can freeze and fail to respond to any environmental stimulus. Enumerative algorithms were proposed in [50] to detect the violation of the stability property for pervasive computing systems. Also, our previous work [10] supports dynamically checking pervasive computing systems' stability at runtime.

Energy efficiency. Pervasive computing systems sense environments by sensors and connect to a tremendous world of users and things. RFID systems and smartphone applications are two typical

examples. However, the services of many real-world pervasive computing systems are realized in an energy-inefficient way [12,18,54], wasting a lot of energy and causing bad user experiences. For example, existing studies found that a large percentage of smartphone applications on market contains various bugs that would cause serious energy waste [12]. To improve the energy inefficiency of pervasive computing systems, many approaches have been proposed recently [18,30,54]. For example, our earlier work studies an interesting type of energy inefficiency problem in pervasive computing systems: sensory data underutilization [18,54] and proposes a systematic analysis approach to diagnosing these problems. This approach conducts code analysis to simulate the runtime behavior of a system. It checks how sensory data are used at each explored system state to locate those system states where sensory data are comparatively underutilized, which may cause energy waste. Other researchers also worked on improving energy efficiency for pervasive computing systems from different perspectives. Some proposed design strategies to reduce energy consumption of pervasive computing systems [53,55]. Others studied energy problems from the security aspect. For instance, Kim et al. proposed to use power signatures based on system hardware states to detect energy-greedy malwares [56].

Performance. Recent studies have also shown that lots of pervasive computing systems suffer from performance inefficiency [12,54]. Performance is a vital property in pervasive computing systems. Bad performance could slow down these systems and affect user experiences. In particular, most pervasive computing systems, for example, smartphones and wearable devices, constantly interact with users. This makes user experiences extremely important in evaluating the service quality of these systems. A recent study collected and analyzed performance problems [12] that commonly occur in pervasive computing systems. The work studied the characteristics (e.g., common types and how problems manifested) and root causes of the collected performance problems and proposed diagnosis algorithms for problem detection.

Atomicity. Nowadays, multithreaded programs are everywhere and also benefited pervasive computing systems (e.g., improving system performance). However, despite the benefits, multithreaded programs also bring many concurrency bugs into pervasive computing systems [57,58]. It is well known that concurrency bugs are notoriously difficult to detect because there can be vast combinations of interleavings among concurrent threads, and yet only a small fraction of them can reveal bugs [59]. What is worse, in pervasive computing systems, context collecting devices (i.e., sensors) may not have synchronized clocks and may run at different speeds, which makes concurrency bug detection even more challenging. Atomicity is an important property in multithreaded programs that guarantees a set of important operations to be executed without any disruption. Recently, a number of serializability criteria have been proposed, including atomicity, causal atomicity, conflict/view serializability, and atomic-set serializability [60]. Our earlier work also proposed a testing technique that can effectively detect atomic-set serializability violations in pervasive computing systems [59]. Still, there are many atomicity-related problems that have not been adequately considered for pervasive computing systems and require more attention.

3.2 Detecting techniques

Different faults have different characteristics. To detect the above discussed faults in pervasive computing systems, researchers have proposed various techniques. In this subsection, we discuss some of them in detail.

Statistical analysis-based detection. Some faults are difficult to recognize, for example, energy inefficiency faults. There are no precise criteria that can be used to judge their existence. Fortunately, these faults typically exhibit some statistical characteristics, which can be used to identify them. Statistical analysis-based fault detection techniques have been popularly studied in recent years. In our previous work [18,54], we proposed a technique to detect an important type of energy inefficiency fault: sensory data underutilization. In pervasive computing systems, sensory data are acquired at the cost of energy and should be effectively used by a system. Sensory data are underutilized when their energy cost outweighs their actual uses. Such underutilization often suggests design or implementation faults that could cause energy waste. To analyze how sensory data are used in a system, one needs to know which program data depend on sensory data and how they are used. This requires analyzing data usage statistics at runtime. Our technique tracks the usage of these data at a bytecode instruction level and adapts dynamic

tainting for a precise analysis of sensory data utilization. For energy analysis, we further designed our data utilization coefficient metric that helps compare data usage across different system states to identify sensory data underutilization cases.

Pattern matching-based detection. Lots of existing studies show that faults in pervasive computing systems demonstrate regular patterns [10–12,50,61], which can be used for fault detection. Sama identified various fault patterns in pervasive computing systems, for example, determinism, stability, liveness, and reachability [11,50]. Similar patterns were also studied in our previous work [10]. Also, we identified a new pattern of consistency faults. The faults violate the requirement that a system's understanding to its environment should be consistent with its actual environmental conditions. These above discussed functional fault patterns have been reported in various failure reports in pervasive computing systems and help enable the automatic fault detection. On the other hand, we also investigated many pervasive computing systems to identify faults that relate to nonfunctional properties of these systems in our earlier work [12]. Specifically, we studied common patterns of performance faults in popular pervasive computing systems and learned how these faults manifest themselves. We further proposed a tool to detect such performance inefficiency faults.

Coverage criteria-based detection. Pervasive computing systems differ from conventional software systems in many aspects. For instance, they are context-aware and may have some special programming paradigms [62]. Due to this reason, most conventional testing and fault detection techniques are not directly applicable [63,64]. For instance, Lu et al. disclosed that conventional data-flow testing techniques are inadequate for pervasive computing systems [63]. To address this problem, they proposed a novel family of context-related data flow coverage criteria to measure the comprehensiveness of test suites for pervasive computing systems. On the other hand, pervasive computing systems may also have some special programming paradigms. As discussed in [62], NesC is a programming language for pervasive computing systems that run on top of networked sensor nodes. Such a system mainly uses interrupt mechanism to trigger a sequence of operations. However, the magnitude of inter-interrupt interleavings in such systems may cause many failures. Existing testing techniques for traditional concurrency programs that only contain a small scale of inter-interrupt interleavings are not applicable for nesC systems. While nesC blocks new interrupt interleavings when handling interrupts, this feature significantly restricts the scale of inter-interrupt interleavings that can occur in a nesC system. Based on this observation, Lai et al. [62] modeled how operations may interleave as inter-interrupt flow graphs, and proposed two test adequacy criteria, one on inter-interrupt data-flows and the other on inter-interrupt control-flows.

Simulation based detection. Simulation is often adopted to study the design and validate the correctness of pervasive computing systems. This is because the behavior of these systems is highly coupled with their software, hardware, and deployment environments and it is impractical to deploy a system in many different environments for locating potential faults. However, even with simulation, it is not easy to validate pervasive computing systems due to various noises and uncertainties in real environments. For example, radio waves in RFID systems can be affected by many environmental factors such as multipath effects, backing materials, and electromagnetic noises. The effect of these factors on radio waves cannot be precisely simulated. In [13], an effective technique was proposed to construct a mixed-reality simulator for pervasive computing systems (i.e., RFID systems) using iterated learning based on a support vector machine. In our previous work, we also used simulation to verify and detect faults in robot-car pervasive computing systems [7,47,49].

4 Uncertainty handling

Uncertainty poses a big threat to the correctness and dependability of pervasive computing systems, which continually sense and adapt to their changing environments [5]. This problem is gaining a lot of attention in recent years. When a pervasive computing system perceives environmental changes, it will adapt its behavior according to its predefined logics. However, there are many uncertain factors affecting pervasive computing systems and their complex environments [7,20]. For example, the interactions between a system and its environment can be affected unexpectedly by uncertainty [7]. More specifically, when

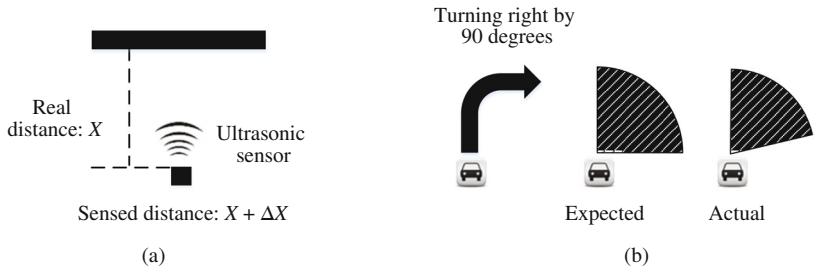

Figure 3 Examples of unreliable environmental sensing and flawed adaptation.

pervasive computing systems sense noisy environments via unreliable sensors, the sensing results will be unfaithful. Besides, pervasive computing systems sometimes react to their environments by uncertain flawed actors. Such uncertainties will impair pervasive computing systems' functionalities if they are not properly handled [47].

While there is no uniform and widely accepted definition of uncertainty in pervasive computing systems, most researchers agree that uncertainty arises from a system state of incomplete or inconsistent knowledge about the environmental or system configurations at a specific point [20]. Many efforts have been made to study and address uncertainty-related problems [65–68]. Ramirez et al. [20] reported a taxonomy of uncertain factors that can affect pervasive computing systems. The uncertainty is classified according to whether a source of uncertainty arises predominantly at requirement, design, or runtime levels. In this survey, we mainly focus on uncertainty at the runtime level according to this classification. Sources of runtime uncertainty occur primarily from interactions between a pervasive computing system and its unpredictable environment [7,20].

In our recent work [7], we modeled the uncertainty caused by unreliable environmental sensing and flawed adaptation in verifying pervasive computing systems. For environmental sensing, a pervasive computing system may only be able to obtain an estimate of its environmental conditions, but never know its real state. As shown in Figure 3(a), an ultrasonic sensor's sensing always contains unpredictable noise, and its sensed values may not faithfully reflect real distances. For adaptation, a pervasive computing system can only interact with its environment as designed, but may not know whether the interaction indeed proceeds as expected. For instance, when a robot-car system controls its robot-car to take a right-turning action, the system's expectation is that the robot-car should turn right by 90°. However, in reality, the car may just turn right for 85° due to its hardware deficiency, as illustrated in Figure 3(b), while the robot-car system may not be able to know whether or to what extent the uncertainty has happened. Such uncertainty can cause pervasive computing systems to fall into failures due to their imprecise or even incorrect understanding to environments. In our previous work, we studied uncertainty in many real cases and observed that uncertainty caused by unreliable sensing or flawed adaptation demonstrates regular patterns. Specifically, the sensed value of unreliable sensing or the effects of flawed adaptation often fall into an error range, with a distribution associated with the physical characteristics of sensing technologies and actions. So, we modeled the uncertainty by providing error ranges for context variables affected by uncertainty. Furthermore, we ranked the verification results (i.e., counterexamples) according to their likelihoods of occurrence in an environment with uncertainty. Another piece of previous work [69] also presented a ranking strategy to prioritize verification results for pervasive computing systems, but by using an environment model inferred from historical environmental data. The environment model contains rich information, including environmental uncertainty. Despite the same goal, there is one major difference between the two pieces of work [7] and [69]. The former considers uncertainty in the verification process, while the latter uses uncertainty to refine the verification results. These techniques can help developers judge whether uncertainty has been properly handled in their pervasive computing systems. Unhandled uncertainty would probably cause undesirable results, for example, consistency failures, which imply that a system's internal state deviates from its actual external environment. As an additional safeguard, existing work [47,49,70] also proposed a runtime failure handling strategy that enables pervasive computing

systems to automatically recover from consistency failures caused by uncertainty.

There is also a large body of research work focusing on how to handle the uncertainty at requirement and design level for pervasive computing systems. Esfahani et al. [65] described an approach to tackling the challenge of uncertainty by assessing both the positive and negative consequences of uncertainty, and proposed a framework for handling uncertainty in pervasive computing systems [66]. RELAX [71] is a requirement language for pervasive computing systems that helps engineers capture uncertainty in requirement definitions. Ghezzi et al. [67] proposed a framework that supports adaptation to nonfunctional manifestations of uncertainty by relying on alternative or optional functionalities. The framework allows engineers to derive from an initial model of the system to a finite state automaton augmented with probabilities. Famelis et al. [68] designed a methodology to express uncertainty using annotations with well-defined semantics to change a model into a partial model and proposed an approach to reasoning with such models.

5 Future work and open issues

Pervasive computing is a well-studied area, yet still progressing area. Researchers have explored several directions and addressed many issues in recent years. However, to realize the full potential of pervasive computing systems, certain topics need further investigation. Future work can be carried out in different categories, which correspond to different factors and challenges affecting the dependability of pervasive computing systems.

First, regarding the context management, more efficient and effective context consistency checking and resolution techniques can be exploited, such as using GPU's capacity to accelerate the process of checking. Besides, automated context inconsistency resolution mechanisms are in great need for different application scenarios. Second, the nature of pervasive computing is to blend-in computing devices within the background environment of our everyday lives. However, due to limited system resources and their dynamic nature, pervasive computing systems have been susceptible to many dependability issues. Several fault patterns have been sorted out in existing studies and our survey, but there still exist more other complex issues that need to be explored and studied in depth, for example, security and energy issues. Lastly, uncertainty raises many challenges to the dependability of pervasive computing systems. The uncertainty can appear at different phases of a pervasive computing system's life cycle, such as requirement and design phases, and even at runtime. What is more, uncertain environmental interactions between a pervasive computing system and its environment pose stiff challenges to its design, test, verification, and validation.

6 Conclusion

In this article, we surveyed the research efforts that have been made to improve the dependability of pervasive computing systems. We presented a taxonomy to organize improvement techniques for pervasive computing systems according to their addressed problems. By following this taxonomy, we discussed three categories of existing research work: context management, fault detection, and uncertainty handling. Each category focuses on one major factor that affects pervasive computing systems' dependability. For example, some techniques consider context an important concern and provide various context inconsistency detection and resolution methods. Some present techniques for fault pattern identification and fault detection to improve pervasive computing systems' dependability. Uncertainty is also taken into account for its significant influence on pervasive computing systems. There are some techniques proposed to handle uncertainty caused by environmental interactions between pervasive computing systems and their environments. Even though most of these discussed techniques have succeeded in improving pervasive computing systems' dependability, they addressed the dependability problem only from one single aspect. In practice, developing and deploying highly dependable pervasive computing systems still face unique challenges. So, we wish that our survey can inspire more efforts not only from research communities, but

also from industries. In this way, we can make solid progress toward dependable pervasive computing.

Acknowledgements

This work was supported by National Basic Research Program of China (973 Program) (Grant No. 2015CB352202), National Natural Science Foundation of China (Grant Nos. 61472174, 91318301, 61321491), and Research Grants Council (General Research Fund 611813) of Hong Kong.

References

1 Weiser M. The computer for the 21st century. Sci Amer, 1991, 265: 94–104
2 Orwat C, Graefe A, Faulwasser T. Towards pervasive computing in health care–Ca literature review. BMC Med Inf Decis Making, 2008, 8–26
3 Judd G, Steenkiste P. Providing contextual information to pervasive computing applications. In: Proceedings of the 2003 IEEE International Conference on Pervasive Computing and Communications, Texas, 2003. 133–142
4 Satyanarayanan M. Pervasive computing: vision and challenges. Personal Commun, 2001, 8: 10–17
5 Cheng B H C, de Lemos R, Giese H, et al. Software engineering for self-adaptive systems: a research roadmap. In: de Lemos R, Giese H, Müller H A, et al., eds. Software Engineering for Self-adaptive Systems. Berlin/Heidelberg: Springer, 2009. 1–26
6 Xu C, Cheung S C. Inconsistency detection and resolution for context-aware middleware support. In: Proceedings of the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Lisbon, 2005. 336–345
7 Yang W, Xu C, Liu Y, et al. Verifying self-adaptive applications suffering uncertainty. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, Vasteras, 2014. 199–210
8 Xu C, Cheung S C, Chan W K. Incremental consistency checking for pervasive context. In: Proceedings of the 28th International Conference on Software Engineering, Shanghai, 2006. 292–301
9 Xu C, Cheung S C, Chan W K, et al. Partial constraint checking for context consistency. ACM Trans Softw Eng Methodol, 2010, 19: 1–61
10 Xu C, Cheung S C, Ma X, et al. ADAM: identifying defects in context-aware adaptation. J Syst Softw, 2012, 85: 2812–2828
11 Sama M, Elbaum S, Raimondi F, et al. Context-aware adaptive applications: fault patterns and their automated identification. IEEE Trans Softw Eng, 2010, 36: 644–661
12 Liu Y, Xu C, Cheung S C. Characterizing and detecting performance bugs for smartphone applications. In: Proceedings of the 32nd International Conference on Software Engineering, Hyderabad, 2014. 1013–1024
13 Cheng H, Liu Y, Cheung S C, et al. Mixed-reality simulation of RFID systems using iterated learning. Int J RF Technol Res Appl, 2012, 3: 219–242
14 Xu X, Gu L, Wang J, et al. Read more with less: an adaptive approach to energy-efficient RFID systems. IEEE J Sel Areas Commun, 2011, 29: 1684–1697
15 Gao C, Wei J, Xu C, et al. Sequential event pattern based design of context-aware adaptive application. Int J Softw Inf, 2010, 4: 419–436
16 Lu A, Fang W, Xu C, et al. Data-driven testing methodology for RFID systems. Front Comput Sci China, 2010, 4: 354–364
17 Wu Y, Chung K K, Qu H, et al. Interactive visual optimization and analysis for RFID benchmarking. IEEE Trans Visual Comput Graph, 2009, 15: 1335–1342
18 Liu Y, Xu C, Cheung S C, et al. GreenDroid: automated diagnosis of energy inefficiency for smartphone applications. IEEE Trans Softw Eng, 2014, 40: 911–940
19 Jeffery S R, Garofalakis M, Franklin M J. Adaptive cleaning for RFID data streams. In: Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, 2006. 163–174
20 Ramirez A J, Jensen A C, Cheng B H C. A taxonomy of uncertainty for dynamically adaptive systems. In: Proceedings of the 2012 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, Zurich, 2012. 99–108
21 Schmidt A, Aidoo K A, Takaluoma A, et al. Advanced interaction in context. Handheld Ubiq Comput, 1999: 89–101
22 Gray P, Salber D. Modelling and using sensed context information in the design of interactive applications. Eng Hum-Comput Interact, 2001, 317–335
23 Harter A, Hopper A, Steggles P, et al. The anatomy of a context-aware application. Wirel Netw, 2002, 8: 187–197
24 Henricksen K, Indulska J, Rakotonirainy A. Modeling context information in pervasive computing systems. Lect Notes Comput Sci, 2002, 2414: 167–180
25 Roman M, Hess C, Cerqueira R, et al. A middleware infrastructure for active spaces. IEEE Pervasive Comput, 2002, 1: 74–83
26 Sousa J P, Garlan D. Aura: an architectural framework for user mobility in ubiquitous computing environments. In:

Bosch J, Gentleman M, Hofmeister C, et al., eds. Software Architecture. New York: Kluwer Academic Publishers, 2002. 29–43

27 Dey A K, Abowd G D, Salber D. A context-based infrastructure for smart environments. In: Paddy N, Gerard L, Simon D, eds. Managing Interactions in Smart Environments. London: Springer, 2000. 114–128

28 Griswold W G, Boyer R, Brown S W, et al. A component architecture for an extensible, highly integrated context-aware computing infrastructure. In: Proceedings of 25th International Conference on Software Engineering, Portland, 2003. 363–372

29 Xu C, Cheung S C, Lo C, et al. Cabot: on the ontology for the middleware support of context-aware pervasive applications. In: Proceedings of the NPC 2004 Workshop on Building Intelligent Sensor Networks, Wuhan, 2004. 568–575

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

31 Nentwich C, Capra L, Emmerich W, et al. Xlinkit: a consistency checking and smart link generation service. ACM Trans Internet Technol, 2002, 2: 151–185

32 Demsky B, Rinard M C. Goal-directed reasoning for specification-based data structure repair. IEEE Trans Softw Eng, 2006, 32: 931–951

33 Xu C, Liu Y P, Cheung S C, et al. Towards context consistency by concurrent checking for Internetware applications. Sci China Inf Sci, 2013, 56: 082105

34 Xu C, Cheung S C, Chan W K. Goal-directed context validation for adaptive ubiquitous systems. In: Proceedings of the 2007 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, Minneapolis, 2007. 7–17

35 Xu C, Cheung S C. Decentralized constraint checking for pervasive computing. In: Proceedings of the 6th Annual IEEE International Conference on Pervasive Computing and Communications, Hong Kong, 2008. 45–48

36 Sui J, Xu C, Wang X, et al. GAIN: GPU-based constraint checking for context consistency. In: Proceedings of 21st IEEE Asia-Pacific Software Engineering Conference, Jeju, 2014. 342–349

37 Xi W, Xu C, Yang W, et al. SHAP: suppressing the detection of inconsistency hazards by pattern learning. In: Proceedings of 21st IEEE Asia-Pacific Software Engineering Conference, Jeju, 2014. 414–421

38 Ranganathan A, Campbell R H. An infrastructure for context-awareness based on first order logic. Personal Ubiq Comput, 2003, 7: 353–364

39 Insuk P, Lee D, Hyun S J. A dynamic context-conflict management scheme for group-aware ubiquitous computing environments. In: Proceedings of the 29th Annual International Computer Software and Applications Conference, Edinburgh, 2005. 359–364

40 Bu Y, Gu T, Tao X, et al. Managing quality of context in pervasive computing. In: Proceedings of the 6th International Conference on Quality Software, Beijing, 2006. 193–200

41 Chomicki J, Lobo J, Naqvi S. Conflict resolution using logic programming. IEEE Trans Knowl Data Eng, 2003, 15: 244–249

42 Xu C, Cheung S C, Chan W K, et al. Heuristics-based strategies for resolving context inconsistencies in pervasive computing applications. In: Proceedings of the 28th International Conference on Distributed Computing Systems, Beijing, 2008. 713–721

43 Xu C, Cheung S C, Chan W K, et al. On impact-oriented automatic resolution of pervasive context inconsistency. In: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Dubrovnik, 2007. 569–572

44 Xu C, Ma X, Cao C, et al. Minimizing the side effect of context inconsistency resolution for ubiquitous computing. In: Proceedings of the 8th ICST International Conference on Mobile and Ubiquitous Systems, Copenhagen, 2011. 285–297

45 Chen C, Ye C, Jacobsen H A. Hybrid context inconsistency resolution for context-aware services. In: Proceedings of 2011 IEEE International Conference on Pervasive Computing and Communications, Seattle, 2011. 10–19

46 Yang H, Xu C, Ma X, et al. ConsView: towards application-specific consistent context views. In: Proceedings of the 36th Annual International Computer Software and Applications Conference, Izmir, 2012. 632–637

47 Xu C, Yang W, Ma X, et al. Environment rematching: toward dependability improvement for self-adaptive applications. In: Proceedings of IEEE/ACM 28th International Conference on Automated Software Engineering, Palo Alto, 2013. 592–597

48 Zhao Y, Cheung S C, Ni L M. LocaToR: locating passive RFID tags with the relative neighborhood graph. In: Proceedings of the IEEE/IFIP 8th International Conference on Embedded and Ubiquitous Computing , Hong Kong, 2010. 154–161

49 Yang W, Xu C, Zhang L. IDEA: improving dependability for self-adaptive applications. In: Proceedings of the 2013 Middleware Doctoral Symposium, Beijing, 2013. 1–6

50 Sama M, Rosenblum D S, Wang Z, et al. Model-based fault detection in context-aware adaptive applications. In: Proceedings of the 16th ACM SIGSOFT International Symposium on the Foundations of Software Engineering, Atlanta, 2008. 261–271

51 Xu C, Cheung S C, Ma X, et al. Dynamic fault detection in context-aware adaptation. In: Proceedings of the 4th Asia-Pacific Symposium on Internetware, Qingdao, 2012. 1–10

52 Xu C, Cheung S C, Ma X, et al. Detecting faults in context-aware adaptation. Int J Softw Inf, 2013, 7: 85–111
53 Hao S, Li D, Halfond W G J, et al. Estimating mobile application energy consumption using program analysis. In: Proceedings of the 35th International Conference on Software engineering, San Francisco, 2013. 92–101
54 Liu Y, Xu C, Cheung S C. Where has my battery gone? Finding sensor related energy black holes in smartphone applications. In: Proceedings of 2013 IEEE International Conference on Pervasive Computing and Communications, San Diego, 2013. 2–10
55 Pathak A, Hu Y C, Zhang M, et al. Fine-grained power modeling for smartphones using system call tracing. In: Proceedings of the 6th European Conference on Computer Systems, Amsterdam, 2011. 153–168
56 Kim H, Smith J, Shin K G. Detecting energy-greedy anomalies and mobile malware variants. In: Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services, Breckenridge, 2008. 239–252
57 Henzinger T A, Jhala R, Majumdar R. Race checking by context inference. In: Proceedings of the 25th ACM SIGPLAN Conference on Programming Language Design and Implementation, Washington, 2004. 1–13
58 Huang Y, Yang Y, Cao J, et al. Runtime detection of the concurrency property in asynchronous pervasive computing environments. IEEE Trans Parall Distrib Syst, 2012, 23: 744–750
59 Lai Z, Cheung S C, Chan W K. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In: Proceedings of the 32nd International Conference on Software Engineering, Cape Town, 2010. 235–244
60 Vaziri M, Tip F, Dolby J. Associating synchronization constraints with data in an object-oriented language. In: Proceedings of the 33rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, 2006. 334–345
61 Xu G, Mitchell N, Arnold M, et al. Finding low-utility data structures. In: Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation, Toronto, 2010. 174–186
62 Lai Z, Cheung S C, Chan W K. Inter-context control-flow and data-flow test adequacy criteria for nesc applications. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Atlanta, 2008. 94–104
63 Lu H, Chan W, Tse T. Testing context-aware middleware-centric programs: a data flow approach and a RFID-based experimentation. In: Proceedings of the 14th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Portland, 2006. 242–252
64 Lu H, Chan W, Tse T. Testing pervasive software in the presence of context inconsistency resolution services. In: Proceedings of the 30th International Conference on Software Engineering, Leipzig, 2008. 61–70
65 Esfahani N, Kouroshfar E, Malek S. Taming uncertainty in self-adaptive software. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, Szeged, 2011. 234–244
66 Esfahani N. A framework for managing uncertainty in self-adaptive software systems. In: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering, Lawrence, 2011. 646–650
67 Ghezzi C, Pinto L S, Spoletini P, et al. Managing non-functional uncertainty via model-driven adaptivity. In: Proceedings of the 2013 International Conference on Software Engineering, San Francisco, 2013. 33–42
68 Famelis M, Salay R, Chechik M. Partial models: towards modeling and reasoning with uncertainty. In: Proceedings of the 2012 International Conference on Software Engineering, Zurich, 2012. 573–583
69 Liu Y, Xu C, Cheung S C. AFChecker: effective model checking for context-aware adaptive applications. J Syst Softw, 2013, 86: 854–867
70 Zhang L, Xu C, Ma X, et al. Resynchronizing model-based self-adaptive systems with environments. In: Proceedings of 19th IEEE Asia-Pacific Software Engineering Conference, Hong Kong, 2012. 184–193
71 Cheng B H C, Sawyer P, Bencomo N, et al. A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty. In: Schürr A, Selic B, eds. Model Driven Engineering Languages and Systems. Berlin/Heidelberg: Springer, 2009. 468–483