Information and Software Technology 74 (2016) 219–229

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Suppressing detection of inconsistency hazards with pattern learning

Wang Xi [a,b], Chang Xu [a,b,*], Wenhua Yang [a,b], Xiaoxing Ma [a,b], Ping Yu [a,b], Jian Lu [a,b]

[a] State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
[b] Department of Computer Science and Technology, Nanjing University, Nanjing, China

ARTICLE INFO

Article history:
Received 19 March 2015
Revised 12 August 2015
Accepted 19 August 2015
Available online 28 August 2015

Keywords:
Context inconsistency
Inconsistency hazard
Pattern learning

ABSTRACT

Context: Inconsistency detection and resolution is critical for context-aware applications to ensure their normal execution. Contexts, which refer to pieces of environmental information used by applications, are checked against consistency constraints for potential errors. However, not all detected inconsistencies are caused by real context problems. Instead, they might be triggered by improper checking timing. Such inconsistencies are ephemeral and usually harmless. Their detection and resolution is unnecessary, and may even be detrimental. We name them inconsistency hazards.

Objective: Inconsistency hazards should be prevented from being detected or resolved, but it is not straightforward since their occurrences resemble real inconsistencies. In this article, we present **SHAP**, a pattern-learning based approach to suppressing the detection of such hazards automatically.

Method: Our key insight is that detection of inconsistency hazards is subject to certain patterns of context changes. Although such patterns can be difficult to specify manually, they may be learned effectively with data mining techniques. With these patterns, we can reasonably schedule inconsistency detections.

Results: The experimental results show that **SHAP** can effectively suppress the detection of most inconsistency hazards (over 90%) with negligible overhead.

Conclusions: Comparing with other approaches, our approach can effectively suppress the detection of inconsistency hazards, and at the same time allow real inconsistencies to be detected and resolved timely.

1. Introduction

Context-awareness is one of the most primary requirements of pervasive computing such as smart-spaces, health-care systems and miscellaneous mobile applications. These applications use context information collected from their environment to automatically adjust their behavior and provide smart services for users. Various context sources are available. For example, most smartphones are equipped with more than 10 types of sensors,[1] including GPS sensors, accelerometers, magnetic sensors, and so on. RFID technology is also widely used for tracking people, animals or cargoes. Besides these hardware-based sources, software-based context sources are also common. Indoor location contexts can be derived from WiFi or magnetic readings [25], and acceleration contexts can be used to detect human activities, such as walking, sitting or falling events [26]. Diverse context sources allow applications to be aware of their environmental conditions. However, contexts can themselves be inaccurate due to inevitable measurement errors or improper context reasoning. For example, RFID devices have been reported to be subject to duplicated reads, missing reads and cross reads [15].

Such inaccuracy can lead to the *inconsistency* [29] problem, which means that contexts can be imprecise, incomplete or even conflicting with each other. Context inconsistency is found to be common and may affect applications unexpectedly [23]. Thus context inconsistencies should be detected and resolved in time. One popular approach is to use *consistency constraints* [29] to specify the properties that must hold concerning the context data used by an application. Such constraints can be formulated from physical laws, common senses and other domain-specific rules defined according to certain application requirements. Typically, consistency constraints should be evaluated as soon as application's environment changes for its adaptation timeliness. If any constraint is violated (i.e., evaluated to a truth value **false**, or **false** for short), an inconsistency is said "*detected*" [29], and should be resolved [1,28]. However, this common practice may be subject to numerous false alarms. A significant part of detected inconsistencies may not be caused by inaccurate context data, but by improper detection timings. We illustrate this problem by the example below:

* Corresponding author. Tel.: +86 25 89680919.
E-mail address: changxu@nju.edu.cn, changxu.cn@gmail.com (C. Xu).

[1] http://developer.android.com/guide/topics/sensors/sensors_overview.html

http://dx.doi.org/10.1016/j.infsof.2015.08.003
0950-5849/

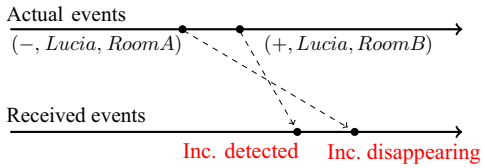

Fig. 1. Example scenario of inconsistency hazards.

A hospital deployed a smart system to help doctors and patients with their daily tasks. This system automatically monitors patients' vital signs and periodically informs their doctors on duty. In case of emergency, it would immediately notify doctors nearby for treatment. Each doctor is attached with an RFID tag for location tracking and carries a smartphone for receiving notifications from the system. Suppose that doctor *Lucia* leaves Room A and then enters Room B. The system's deployed sensors would capture such movement events as "*Lucia disappears from Room A*" and "*Lucia appears in Room B*" to update the system's context information. However, sensors may fail to capture some events due to noises, say, missing the "*disappearing*" event. This would make the system think of *Lucia* appearing in both rooms at the same time, which clearly violates physical laws. An inconsistency would result due to this violation and should be resolved before contexts involved in inconsistency (named *inconsistent contexts*) are used by the system.

In the real world, a single event can trigger a group of related changes to context information, and these changes may be sensed and reported by different context sources with different update rates. If a consistency constraint is evaluated when only part of these changes have taken effect while others have not yet, this constraint may behave as being violated, leading to detection of an inconsistency. However, this inconsistency can be merely a false alarm, as later it would be gone spontaneously after other related changes are applied. In this case, this detected inconsistency does not indicate real context problems, but instead is transiently caused by scheduling inconsistency detection when not all contexts are ready. Such inconsistencies do not require resolution, and if resolved, they may cause unexpected consequences instead. We name such false alarms *inconsistency hazards*, which conceptually resemble hazards in digital circuits [21].

Consider our earlier example. Suppose that the location sensor installed in Room A updates at a lower rate (say, 10 s) than the one in Room B (say, 8 s). It can be the case that some changes (e.g., event "*Lucia appears in Room B*") are received and then applied earlier than other related changes (e.g., event "*Lucia disappears in Room A*"). If inconsistency detection is right scheduled between these two batches of changes (i.e., related changes are isolated), an inconsistency hazard would be detected, as shown in Fig. 1. If this hazard is treated as a real inconsistency, it may be resolved by removing the existence of *Lucia* from Room B to avoid violating physical laws. As a result, her latest location information would be missed due to this false alarm.

We observe that inconsistency hazards can occupy a significant proportion of all detected inconsistencies, ranging from 8.1% to 62.2% in our investigated three context-aware applications (discussed later in evaluation). Detecting and resolving these hazards as real inconsistencies can waste valuable computing resources, which should instead be used for other application functionalities. Besides, some context information, which is actually valid, might be wrongly updated or deleted to resolve these hazards, and thus affect the execution of concerned applications. Hence, inconsistency hazards should be recognized or their detection should be suppressed, and this should preferably be done in an automated way.

However, it is not easy to tell whether a detected inconsistency is a real inconsistency or a hazard as they are both caused by violation of consistency constraints. In this article, we aim to address this problem by suppressing detection of inconsistency hazards by learned patterns. Our key observation is that constraint checking is subject to inconsistency hazard only under certain patterns of context changes. These patterns might not be easily specified manually, but can be effectively learned from historical inconsistency detection data with data mining techniques. One can use this knowledge to schedule inconsistency detection to effectively suppress potential hazards. We name our approach **SHAP**, which stands for *Suppressing Inconsistency HAzards with Pattern-Learning*. **SHAP** was initially reported in [24], and in this article we extend it and proposed a new strategy named **SHAP**$^{+}$, which is more effective in suppressing inconsistency hazards with very short delay. The new strategy is compared with existing strategies on our experimental subjects, and more results have been presented.

The remainder of this article is organized as follows. Section 2 introduces background knowledge about context inconsistency detection and Section 3 further explains our inconsistency hazard problem. Sections 4 and 5 elaborate on our approach for hazard suppression, with focus on detection scheduling and pattern learning, respectively. Section 6 evaluates our **SHAP** approach experimentally. Section 7 discusses related work, and finally Section 8 concludes this article.

2. Background

In this section, we introduce background concepts concerning context inconsistency detection.

2.1. Context modeling

A context refers to a piece of information that can be used to characterize the situation of an entity [5], and context modeling explains how contexts are represented. Various context modeling techniques have been proposed [16]. In this article we model a *context* as a finite set of associated elements, each of which specifies one aspect concerning its targeted entity. An element can have several *fields*, and each field contains a numerical or textual value. For example, the current status of a doctor (e.g., {**location**: *Ward*3824, **name**: *Lucia*, $\cdots$}) can be such an element. Then, all such elements can compose a context *DOCT*, representing all doctors currently in this hospital. An application can use various contexts. We use a *context pool* to collect all contexts interesting to an application. Besides *DOCT*, the pool can also contain other contexts about indoor environmental information (e.g., temperature, humidity, etc.) and conditions of patients in each ward of this hospital.

By definition, a context naturally supports three operations: **adding** a new element into, **deleting** an existing element from, or **updating** an existing element in a context. We name these operations *context changes*. A context change is modeled as a tuple (**t, c, e**), where **t** represents the type of the change (i.e., **add, delete** or **update**) and **c** represents the context this change is to be applied to. The concrete element to be affected (i.e., added, deleted or updated) is represented by **e**.

We note that our model is suitable for both environmental contexts and logical contexts. There is no difference between the treatments for different types of contexts.

2.2. Inconsistency detection

Consistency is an important property for computer systems, such as distributed systems [4] and database systems [7]. Contexts used by context-aware applications are also obliged to consistency. We check contexts against pre-specified consistency constraints to ensure consistency [29]. These constraints can be expressed using the following first-order logic based language:

$$f := \forall e \in C(f) | \exists e \in C(f) | (f) \wedge (f) | (f) \vee (f) \\ | (f) \rightarrow (f) | \neg (f) | bfunc(param, \cdots, param). \quad (1)$$