Suppressing Detection of Inconsistency Hazards with Pattern Learning

Wang Xi, Chang Xu*, Wenhua Yang, Xiaoxing Ma, Ping Yu , Jian Lu

State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China

Department of Computer Science and Technology, Nanjing University, Nanjing, China

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.

Keywords: context inconsistency, inconsistency hazard, pattern learning

1. Introduction

Context-awareness is one of the most primary requirements of pervasive computing such as smart-spaces, health-care systems and miscellaneous mobile

*Corresponding author
Email address: changxu@nju.edu.cn (Chang Xu)

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 ten 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:

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.

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

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 seconds) than the one in Room B (say, 8 seconds). 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 Figure 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 cer-

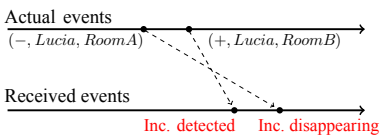


Figure 1: Example scenario of inconsistency hazards

tain 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., $\{\textbf{location} : Ward3824, \textbf{name} : Lucia, \cdots\}$) 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, ...) 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 prespecified consistency constraints to ensure consistency [29]. These constraints can be expressed using the following first-order logic based language:

$$\begin{aligned} f \quad := \quad & \forall e \in C(f) | \exists e \in C(f) | (f) \wedge (f) | (f) \vee (f) \\ & | (f) \rightarrow (f) | \neg (f) | bfunc(param, \cdots, param). \end{aligned} \tag{1}$$

In the above syntax, symbol C represents a *context* and variable e can take any *element* from C as its value. Terminal *bfunc* represents any domain- or application- specific function that returns **true** or **false** based on values of its parameters, e.g., $equals(str_1, str_2)$ or $greater(v_1, v_2)$. Parameters of *bfunc* can be fields of elements or other constants. Consider a consistency constraint *"nobody can be in two rooms (A and B) simultaneously"*. It can be expressed as follows:

$$\forall e_1 \in R_A(\neg(\exists e_2 \in R_B(equals(e_1.id, e_2.id)))). \tag{2}$$

R_A and R_B are two contexts representing people currently staying in Room A and Room B, respectively. Function *equals* checks whether two string variables are identical. The common practice is to evaluate a constraint immediately after any related context (e.g., R_A and R_B for this constraint) is changed. We name this practice Traditional approach (or **Trad** for short).

3. Inconsistency Hazard

Many context inconsistencies detected by **Trad** disappear shortly after their detection. Such ephemeral inconsistencies do not indicate real context problems as we explained earlier. They are essentially inconsistency hazards.

The underlying cause of inconsistency hazard is scheduling inconsistency detection in the middle of a group of related context changes. This can be caused by misalignment of different context sources with various updating rates, as we illustrated in Figure 1. However, even if context changes are from the same source, hazards may still occur because there is no explicit boundary for related context changes. Let us revisit the aforementioned hospital example. Suppose that a consistency constraint requires any ICU in the hospital having any patient inside must have at least one doctor watching on her status continuously. This

constraint can be formally specified as follows (ICU_x represents all patients currently detected in ICU X):

$$(\exists p \in ICU_x(isTrue(p.isPatient))) \rightarrow (\exists d \in DOCT(equals(d.location, \text{“}ICU_x\text{”}))). \tag{3}$$

Suppose that a doctor *Lucia* accompanies a patient *Jenny* into ICU X. The entering events are captured as two context changes, chg_1 and chg_2, which are to **update** *Lucia*'s location to "ICU_x" in context $DOCT$ and **add** *Jenny*'s information into context ICU_x, respectively. If *Jenny* happens to sit in a wheelchair and be pushed into the ICU by *Lucia*, context change chg_1 would probably be captured after chg_2. As a result, a context inconsistency would be reported by **Trad** (when chg_2 is handled), but it is a hazard as it would disappear when chg_1 is later captured and handled.

However, **Trad** does not know that this inconsistency is a hazard and will resolve it as usual. There are various resolution strategies [1, 28]. One resolution strategy may consider context change chg_2 unreliable and choose to discard it. Then, although this inconsistency is seemingly resolved, the application would thus miss *Jenny*'s information as well as failing to operate medical devices for her in the ICU. Or, another resolution strategy may keep chg_2, but make the application believe that a patient in the ICU is being unattended. As a result, warning notifications would be sent to nearby doctors and such false warnings can be annoying.

Therefore, resolving inconsistency hazards as real inconsistencies is unnecessary and may even be harmful. We in this article choose to suppress detection of such inconsistency hazards as applications' timeliness requires. One characteristic of hazards is that they are not persistent, i.e., they will disappear after a small set of later context changes are applied. The size of this set is a threshold (denoted as ST), which can be application-specific. In this article, we are interested in how one can predict whether a context inconsistency is hazard without actually detecting it when facing context changes to be handled.

4. Suppressing Inconsistency Hazards

In this section, we present our approach to suppressing the detection of inconsistency hazards. As we have discussed, inconsistency hazards do not persist. So the most intuitive way is to wait and remove all transient inconsistencies. This approach is named Delay-based approach (or **Del** for short). **Del** evaluates consistency constraints as **Trad** does, except that any detected inconsistency would not be reported or resolved until it has survived the duration of the next N context changes. Here, N is user-customizable and ranges from 1 to ST, inclusively. It is clear that **Del** can preserve the detection of all real inconsistencies and effectively remove all hazards if N is set to ST exactly. However, this is achieved at the cost of delaying reporting real inconsistencies up to ST

changes. Although N can be set to a smaller value to shorten the delay, more hazards will result accordingly. So this approach is only suitable for scenarios where the timeliness of resolving real inconsistencies is not emergent but no hazards can be tolerated.

Another approach is to conduct inconsistency detection every N context changes ($N \leq$ ST). This approach is named Batch-based approach (or **Bat** for short). **Bat** can alleviate the hazard problem by suppressing part of hazards. Its delay of reporting detected inconsistencies varies between 0 and N ($N/2$ on average). However, some hazards can still be detected, and their ratio cannot be controlled. **Bat**'s advantage is that the scheduling of inconsistency detections can be greatly reduced, and it is thus suitable for scenarios where computing resources are very limited.

These two approaches are acceptable in some cases, and straightforward to implement. However, neither of them can effectively suppress the detection of inconsistency hazards, and at the same time allow real inconsistencies to be detected and resolved promptly. To achieve this goal, we need to further investigate the hazard problem.

4.1. Hazard Pattern

We observe that hazards for a constraint always occur as a result of some certain sequences of context changes. We define such context change sequences as **hazard patterns**. Hazard patterns are usually made up of a group of related context changes triggered by some real-world events. Detecting inconsistencies in the middle of such sequences may incur inconsistency hazards.

Consider our earlier constraint specified in Equation (3). When several patients enter an empty ICU, followed by a doctor who accompanies these patient(s), a group of context changes will be triggered. There will be several adding changes, one for each patient, and one updating change for the doctor. If we conduct inconsistency detection in the middle of these changes, hazards, which are going to disappear after the doctor's location is finally updated, would be detected. So, one possible hazard pattern for this constraint is

$$[(+, ICU, p_1), (+, ICU, p_2), ..., (\#, Doct, d_x)]$$

Here we use $+$, $-$ and $\#$ to denote three context change types **add**, **delete** and **update**, respectively. With the knowledge of such hazard patterns for each constraint, one can automatically schedule inconsistency detection to suppress hazards. Some hazard patterns are universal and obvious. For example, a string of context changes with the same type (add or delete) and referring to the same context is very likely to be a hazard pattern, since it indicates that this very context may not be ready yet. However, it is not straightforward to find out other patterns. Besides, different constraints usually have different hazard patterns, depending on their semantics and the contexts involved. We adopt classification techniques adapted from the data mining area to recognize hazard patterns and distinguish them from other context change sequences. Classification details will be discussed later in Section 5. In the remainder of

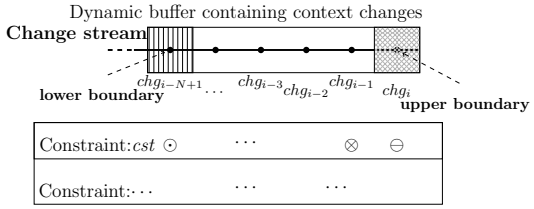


Figure 2: Dynamic buffer for scheduling inconsistency detection

this section, we will elaborate on our **SHAP** approach with the assumption that hazard patterns have been differentiated from normal change sequences by our classifiers.

4.2. SHAP *Approach*

SHAP uses hazard patterns to schedule inconsistency detection smartly. In the following, we first restrict hazard patterns to context change pairs. Then, we extend hazard patterns to an arbitrary number of context changes as a sequence. Consider a constraint *cst* and a context change *chg*. If *chg* does not affect *cst* at all (i.e., *cst* does not contain any sub-formula referencing to *chg*'s concerned context), *cst* does not have to be evaluated. Otherwise *cst*'s truth value may change and it is subject to checking. In this case, inconsistency detection for *chg* will be scheduled by **SHAP**.

4.2.1. Schedule with Context Change Pairs

We use a dynamic buffer to support our scheduling process, as illustrated in Figure 2. Here, *cst* is a consistency constraint under consideration, and chg_i represents the ith received context change. The size of this buffer (denoted by N) will affect the effectiveness of inconsistency detection, and is thus user-customizable according to application requirements. Generally, increasing the size of the buffer would suppress inconsistency hazards more effectively, but at the same time, introducing longer delay in reporting real inconsistencies. Cells under these changes indicate whether inconsistency detection should be scheduled after receiving these changes. Their values can be $\odot$, $\otimes$ or $\ominus$, representing "yes", "no" or "undecided yet", respectively.

We propose two scheduling strategies, named **SHAP-Min** and **SHAP-Max**, respectively. **SHAP-Min** tries to minimize the loss of detected real inconsistencies, while **SHAP-Max** tries to maximize the suppression of detected inconsistency hazards. Algorithm 1 shows how **SHAP-Min** makes scheduling decisions when a new context change chg_i is received. Constraints are considered individually. **SHAP-Min** first schedule chg_i as $\otimes$ if chg_i does not affect a constraint *cst* at all (Lines 2-4). Otherwise, it needs further consideration and thus is temporarily scheduled to $\ominus$ (Line 6). Then chg_i is used to update scheduling decisions for earlier changes if their decisions have not been decided yet (Lines 7-11). Finally, scheduling decision for the earliest change in the buffer

will be set to $\odot$, if it still remains to be $\ominus$ (Lines 13-15), since it has already reached the lower boundary of the buffer.The *isHazardPattern* function used at Line 8 takes two context changes as input and returns **true** or **false** based on our classification result, to be explained later.

SHAP-Min can suppress the detection of most hazards, but still some may escape. Consider a change pair $[chg_i, chg_j]$ $(i < j)$ that is recognized as a hazard pattern. Then inconsistency detection will not be scheduled for chg_i. However, some other change chg_k between them may be scheduled for inconsistency detection if no future change can form any hazard pattern with it. Then inconsistency detection will still be conducted when chg_k is applied (after chg_i but before chg_j). As a result, hazards may occur due to this scheduling (essentially caused by chg_i). We thus propose another scheduling strategy **SHAP-Max** to address this problem. In **SHAP-Max**, when a change pair $[chg_i, chg_j]$ is classified as a hazard pattern, the scheduling decision of inconsistency detection for any change between this pair (including chg_i but excluding chg_j) will be set to $\otimes$. This strategy can suppress more hazards, but might miss few real inconsistencies. We omit the complete description of this strategy's algorithm due to its similarity to **SHAP-Min**, but we note that the only difference lies at Line 9, where **SHAP-Max** will schedule all changes between chg and chg_i as $\otimes$.

We give the whole **SHAP** inconsistency detection process by Algorithm 2. When a new context change chg_i is received, it is first used to update scheduling decisions for all buffered changes. The *updateScheduling* function at Line 1 can be **SHAP-Min** or **SHAP-Max**. After that, inconsistency detection will be conducted if necessary according to updated scheduling decisions.

4.2.2. Schedule with More Context Changes

Hazard patterns are not necessarily context change pairs. Involving more context changes in a hazard pattern may improve the effectiveness of our **SHAP** approach but also make it more complicated. Algorithm 3 presents the process of scheduling inconsistency detection with hazard patterns containing arbitrary numbers of context changes (i.e., can be more than two), and we name it **SHAP**$^+$. The input and output of this algorithm are the same as **SHAP-Min** or **SHAP-Max**, and similarly we maintain a dynamic buffer whose size is still customizable by users. **SHAP**$^+$ first decides whether the currently retained context changes in the dynamic buffer (as a sequence) is a hazard pattern for a constraint (Lines 2-4). If yes, we know that some hazards are likely to be detected, and therefore one should schedule $\otimes$ to all changes in the buffer as **SHAP-Max** does. Besides, the latest received change chg_i should be scheduled as $\ominus$ for further consideration (Lines 4-9). Otherwise, chg_i will be scheduled according to whether it affects this constraint or not (Lines 10-17). Finally, the context change at the lower boundary of the buffer will be scheduled as $\odot$ if its scheduling decision still remains to be $\ominus$ (Lines 18-20). The function *isHazard-Pattern* at Line 4 takes a context change sequence as input, and decides whether this sequence is a hazard pattern or not. Different from the **SHAP** algorithms, the **SHAP**$^+$ algorithm's input is a sequence of changes.

```
Input: chg_i (new context change), Csts (all constraints), buf (dynamic
       buffer of context changes), sch (scheduling decisions), N (size of
       the dynamic buffer)
Output: sch (updated scheduling decisions for each constraint)
1  foreach cst in Csts do
2      if !cst.affectedBy(chg_i) then
3          sch(cst, chg_i) := ⊗
4      end
5      else
6          sch(cst, chg_i) := ⊖;
7          foreach chg in buf do
8              if sch(cst, chg) = ⊖ ∧ cst.isHazardPattern(chg, chg_i) then
9                  sch(cst, chg) := ⊗
10             end
11         end
12     end
13     if buf.size() ≥ N ∧ sch(cst, buf.getTop()) = ⊖ then
14         sch(cst, buf.getTop()) := ⊙
15     end
16 end
```

Algorithm 1: SHAP-Min scheduling strategy

SHAP$^+$ is more complicated to implement, as we will discuss in the next section. However, we note that one only need to conduct classification at most once for each constraint when a new context change is received, and our later experimental results show that **SHAP**$^+$ can be more effective in suppressing inconsistency hazards than **SHAP**, especially when the buffer size is relatively small.

5. Deriving Hazard Patterns

In the previous section, we assumed that we have classifiers that can differentiate hazard patterns from normal context change sequences. In this section, we introduce the classification details. We use **Weka** [8] as our toolkit to train classifiers from historical inconsistency detection data. To train classifiers, we first need to represent context changes in a unified format readable to **Weka**. Then we need context change pairs or sequences to be labeled as *safe patterns* or *hazard patterns* in an automated way. Finally, we train classifiers for detection scheduling purposes.

5.1. Representing Context Changes

The normal data format of **Weka** (named *propositional format*) is a collection of instances sharing a set of features. Features for a context change include

Input: chg_i (new context change), $Csts$ (all constraints), buf (dynamic buffer of context changes), sch (scheduling decisions), N (size of the dynamic buffer), $contextPool$ (all contexts)
Output: $incs$ (detected inconsistencies)

```
1  updateScheduling(chg_i);
2  buf.append(chg_i);
3  if buf.size() ≥ N then
4  |  chg' := buf.top();
5  |  buf.pop();
6  |  ctx := contextPool.get(chg'.context);
7  |  ctx.apply(chg');
8  |  foreach cst in Csts do
9  |  |  if sch(cst, chg_i) = ⊙ then
10 |  |  |  evaluate cst and report detected inconsistencies incs
11 |  |  end
12 |  end
13 end
```

Algorithm 2: SHAP inconsistency detection

its *type*, concerned *context* and concrete *element value* it contains. Unfortunately, contexts from different sources are usually heterogeneous (e.g., elements from two different contexts can have different sets of fields). Contexts have to share the same structure for classifier training purposes. Therefore, we require that each context should be associated with a piece of meta-information, as illustrated in Figure 3. We can then use a database **join** operation to merge multiple contexts' meta-information. Accordingly, the global meta-information of a context pool would be the join result of all its collected contexts' meta-information.

For example, suppose that context ICU_x contains two fields: **name** (a string literal) and **isPatient** (a boolean value), and context $DOCT$ contains three fields as illustrated in Figure 3, i.e., name, location and age. Then their join result concerns four fields in its meta-information, i.e., **name**, **location**, **age** and **isPatient** (the order is not important). Consider a context change $(add, ICUx, \{\textbf{name}{:}Jenny, \textbf{isPatient}{:}\textbf{true}\})$. It will be converted into an instance as follows: $(add, ICU_x, Jenny, ?, ?, \textbf{true})$, where "?" means no value for this instance. The first two features of this instance represent its *type* and *context*, and the remaining four correspond to all its non-equivalent fields in concerned contexts.

5.1.1. Representing Context Change Pairs

Weka's propositional format is naturally suitable for representing context change pairs. Each instance here consists of exactly two context changes, and thus can be simply constructed by combining features extracted from these two changes. For example, a context change pair: [(add, ICU_x, {**name**:*Jenny*,

Input: chg_i (new context change), $Csts$ (all constraints), buf (dynamic buffer of context changes), sch (scheduling decisions), N (size of the dynamic buffer)
Output: sch (updated scheduling decisions for each constraint)

```
1  foreach cst in Csts do
2  |  sqs := a list of all context changes in buf;
3  |  sqs.append(chg_i);
4  |  if cst.isHazardPattern(sqs) then
5  |  |  foreach chg in buf do
6  |  |  |  sch(cst, chg) := ⊗
7  |  |  end
8  |  |  sch(cst, chg_i) := ⊖
9  |  end
10 |  else
11 |  |  if !cst.affectedBy(chg_i) then
12 |  |  |  sch(cst, chg_i) := ⊗
13 |  |  end
14 |  |  else
15 |  |  |  sch(cst, chg_i) := ⊖
16 |  |  end
17 |  end
18 |  if buf.size() ≥ N ∧ sch(cst, buf.getTop()) = ⊖ then
19 |  |  sch(cst, buf.getTop()) := ⊙;
20 |  end
21 end
```

Algorithm 3: SHAP$^+$, scheduling with context change sequences

isPatient:**true**}), (*upd*, *DOCT*, {**name**:*Lucia*, **location**:ICU_x, **age**:26})] will be represented as an instance: $(add, ICU_x, Jenny, ?, ?, \textbf{true}, upd, DOCT, Lucia, ICU_x, 26, ?)$.

5.1.2. Representing Context Change Sequences

Normal context change sequences may contain an arbitrary number of context changes. Thus **Weka**'s normal propositional format is no longer suitable. **Weka** provides another classification schema for this case: Multi-instance (MI) classification[2]. In MI, an example contains a sequence id, a class label and a bag of instances. Each example can correspond to a change sequence and each instance in the bag can correspond to a change in the sequence.

[2] http://weka.wikispaces.com/Multi-instance+classification

```
<contexts>
 <context name="DOCT">
  <field name="name" type="string"/>
  <field name="location" type="enum-string">
   <enumerate value="RoomA" />
   <enumerate value="RoomB" />
   <!-- ... -->
  </field>
  <field name="age" type="integer" />
 </context>
<!-- ... -->
</contexts>
```

Figure 3: Context meta-information

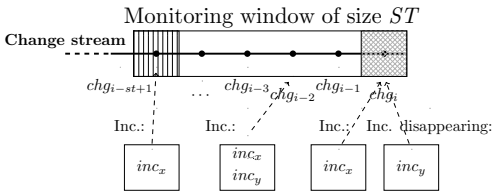


Figure 4: Labeling context change sequences

5.2. Labeling Context Change Sequences

The second step is to label collected context change sequences for classifier training purposes. As illustrated in Figure 4, we keep track of all detected inconsistencies for each constraint caused by recent ST changes with a monitoring window. Here, chg_i is the latest received context change, and chg_{i-ST+1} is the earliest change at the lower boundary of the window. Inconsistencies inc_x and inc_y are detected after chg_{i-ST+1} and chg_{i-2}, respectively. However, inc_y disappears after chg_i, while inc_x is still preserved.

If an earlier inconsistency is detected after applying chg_j $(i-st+1 \leq j < i)$ and disappears immediately after applying chg_i, this inconsistency is considered as a hazard (e.g., inc_y, where $j = i-2$). Accordingly, the changes between chg_j and chg_i will be labeled as *hazard patterns* in our **SHAP** approach. If a change at the lower boundary is finally removed from the window and at that time it has not been labeled into any hazard pattern, then we label the current changes in the buffer with *safe patterns*.

We detect context inconsistencies with **Trad** and label context change sequences for a training set. In this process, the input context changes can be either real changes or simulated changes, as long as the simulated changes satisfy statistical distribution as real changes. To improve the effectiveness of trained classifiers, the raw training data (denoted as T0) needs preprocessing. T0 can be highly imbalanced (e.g., most sequences are labeled as *safe patterns*) and such imbalanced training data can lead to biased classification models [14]. So the first step is to randomly sample a relatively small subset T1 from T0. Then we expand T1 by adding all prefixes of each sequence in T1, with their labels

set according to original sequences in T0. Thus **SHAP** can have the change of recognizing hazard patterns with fewer context changes and suppressing inconsistency hazards as early as possible. Finally, all conflicting (with exactly the same features but different class labels) sequences are all labeled as *hazard pattern*s to resolve the tie for safety. Generally, with more raw data obtained, we can train better classifiers. We use ten-fold cross validation [8] to measure the accuracy of the trained classifiers. We stop collecting training data when the trained classifiers' accuracy is good enough (e.g., the classifiers' F-measure[3] greater than 0.85).

As discussed earlier, **SHAP** can utilize either context change pairs or context change sequences as hazard patterns. T1 can be directly used as training data for context change sequences. For context change pairs, we can extract a subset of T1 whose instances contain only two context changes, and convert the data in this subset into a new training set T2 of propositional format for training.

5.3. Training Classifiers

Weka provides many popular classification algorithms, i.e., J48, Multilayer Perception, and Naive Bayes [22], for training classifiers from propositional data sets. These algorithms can be applied on T2. All **Weka**'s classifiers contain two methods: *buildClassifier*() and *distributionForInstance*(). The former is used to build classification models based on labeled instances, and the latter returns an array (denoted as *dist*) of probabilities, representing the class distribution for a given new instance. The *isHazardPattern* function used in our **SHAP** algorithms converts a pair of, or a sequence of, context changes into an instance and feeds it to function *distributionForInstance* in **Weka**. Its returned value *dist* is then used to calculate whether this change pair or sequence is a *hazard pattern* (we use 0.5 as the threshold in our **SHAP** approach and experiments) or not.

Weka also provides a set of multi-instance classification algorithms, i.e., MIBoost, and there exist third-party libraries making other classification algorithms available, e.g., Hidden Markov Model[4]. These algorithms can be applied to T1.

Our **SHAP** can evaluate a given training set against a set of pre-specified algorithms to choose the one with the best performance (i.e., highest F-measure). This treatment is suitable in most cases. In rare cases, the selected algorithm may be subject to generalization error when the training data are not so representative. Therefore our **SHAP** also allows its users to specify their preferred algorithms according to their domain knowledge concerning hazard patterns.

Currently, all these classifiers are trained in an offline manner. **SHAP** loads pre-trained classifiers before its runtime inconsistency detection scheduling. Training time depends on the training algorithm used and the size of the training set. For our case, the training sets used in our experiments, which

[3] http://en.wikipedia.org/wiki/F1_score
[4] http://doc.gold.ac.uk/~mas02mg/software/hmmweka/

will be discussed in the next section, contain tens of thousands to millions of instances, and the training process can terminate within several minutes, which is quite efficient.

6. Evaluation

We conducted experiments to evaluate the effectiveness of our hazard suppression approaches. **Trad** is used as the baseline to disclose all real inconsistencies and hazards, as well as their relative ratio. All experiments were conducted on a desktop PC with an Intel(R) Core(TM) i5 CPU@3.2GHz and 2GB RAM. This PC is installed with Ubuntu Linux 12.04 and Oracle Java 8. The **independent variables** of our experiments include the certain approach used in inconsistency detection (**Trad**, **Del**, **Bat**, **SHAP-Min**, **SHAP-Max** and **SHAP**$^+$), and parameter N, which decides the size of the dynamic buffer in our **SHAP** approach. The **dependent variables** include the *effectiveness* and *efficiency* of the conducted inconsistency detection. We measure *effectiveness* by *false positive rate* (percentage of detected hazards against all detected inconsistencies) and *false negative rate* (percentage of missed real inconsistencies against all real inconsistencies). *Efficiency* is measured indirectly by comparing *the total number of constraint evaluations conducted in inconsistency detection*.

6.1. Experimental Subjects

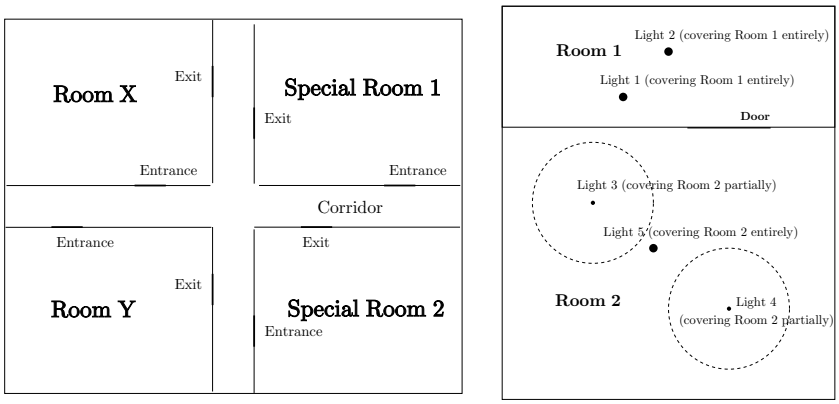
Experiments were conducted through three real-world or simulated context-aware applications. They are:

6.1.1. Smart exhibition application

The smart exhibition application is deployed in our department building to support context-aware exhibition activities. It manages four exhibition rooms and a connecting corridor, as illustrated in Figure 5a. Each room has two doors, and each door is installed with an RFID reader to detect visitor entering or leaving events. The application would customize exhibition contents according to visitors' locations and interests dynamically. Eight consistency constraints are deployed for this application. Six of them are designed to ensure location consistency (Type 1, similar to Equation (2)) and the other two are for ensuring visitors in Special Room 1 and Special Room 2 being accompanied by guides (Type 2, similar to Equation (3)).

6.1.2. Smart light application

The smart light application [20] can dynamically adjust the luminance of lights in a workplace according to locations of workers in the workspace for saving energy, and at the same time meet minimal illumination requirements. For experimental purposes, we simulated a scenario for testing this smart light application as illustrated in Figure 5b. We deployed six consistency constraints according to this scenario's physical layout and laws. Four of these constraints were designed for ensuring inclusion relations. For example, workers covered (in



(a) Smart exhibition application (b) Smart light application

Figure 5: Experimental scenarios

terms of illumination) by Light 1/2/3/4 must also be covered by Light 2/1/5/5. One example is given by the following Equation(4), in which $Light_1$ and $Light_2$ denote contexts for workers covered by these two lights. The other two constraints were designed for ensuring disjoint relations. For example, workers covered by Light 1/3 must not be covered by Light 5/4 (similar to Type 1 given by Equation (2)).

$$(\exists u_1 \in Light_1(True)) \rightarrow (\exists u_2 \in Light_2(equals(u_1.id, u_2.id))). \quad (4)$$

6.1.3. Battery monitor application

This application monitors context information of a smartphone's battery, charger, CPU and screen, and provides suggestions for better power management. The application runs with data collected from UbiComp 2014 programming competition[5], which provides a massive and global dataset of real-world smartphone usage. The application uses four consistency constraints to detect anomalies in data concerning battery and charging status, e.g., if a smartphone's charger is connected, its battery must be in a *charging* state, as given by the following Equation (5). Besides, we are investigating for more constraints concerning other types of sensory data.

$$\forall battery \in power.battery(\exists charger \in power.charger(\\ (equals(charger.state, \text{“}USB\text{”})) \rightarrow (equals(battery.state, \text{“}charging\text{”})))). \quad (5)$$

[5]http://ubicomp.org/ubicomp2014/calls/competition.php

Application	ST	#Constraints	All incs.	Hazard rate
Smart exhibition	7	**Type 1** (6)	26,328	7.57%
		Type 2 (2)	26,304	19.3%
		Total (8)	52,632	15.2%
Smart light	5	**Total** (6)	16,800	62.2%
Battery monitor	6	**Total** (4)	58,586	8.1%

Table 1: Detected hazards (incs.=inconsistencies)

6.2. Detected Hazards

We ran the smart exhibition and smart light applications for 24 hours each. Their collected context changes were 14,973 and 36,850 per hour, respectively. This corresponds to scenarios of a group of 20 visitors accompanied by 5 guides and a workspace containing 5 workers, respectively, which can be considered as typical scenarios. For the Battery monitor application, its context changes were directly derived from UbiComp 2014 data, whose number is 1,912,502 in total. We first detected context inconsistencies for these applications with **Trad** (i.e., detecting inconsistencies eagerly). Table 1 lists the number of detected inconsistencies and their hazard rate for each application subject. We observe that the hazard rate varies from 8.1% to 62.2%, which is significant. Even for the same application (e.g., Smart exhibition application), its hazard rate can be different for different types of constraints (e.g., 7.57% for Type 1 constraints and 19.3% for Type 2 constraints). This imposes challenges to inconsistency detection that should suppress the detection of inconsistency hazards for various constraints.

We observe that the hazard rate can vary greatly with different constraint types and different application scenarios. Examples of constraints with high hazard rates are two constraints in the Smart light scenario: "*any worker covered by Light 1 must also be covered by Light 2*", which is illustrated in Equation (4), and its inverse proposition "*any worker covered by Light 2 must also be covered by Light 1*". Whenever some worker walked from Room 2 to Room 1 (or from Room 1 to Room 2), there must be two related context changes, one for Room 1, and another for Room 2. No matter which change is received first, a hazard would be detected for one of the two constraints. This fact implies an important characteristic of constraints that can easily suffer hazards: containing multiple contexts that are related due to a single event. In future, we plan to further investigate such implied characteristics that can easily incur hazards.

Table 1 also lists ST values, which were set according to application requirements. For example, in the smart exhibition application, its visitors usually gathered into small groups of 5 or 6 people each, and therefore its ST value (indicating how long a hazard can survive) was set to 7.

6.3. Hazard Suppression

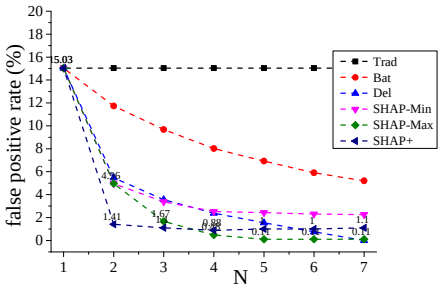
Since the detected hazards are so common, we then evaluated whether our proposed approaches can effectively suppress the detection of these hazards.

Training sets are obtained by the process described in Section 5. We used part of the context data to collect training sets, and then performed inconsistency detection with these trained classifiers on the whole data set. About one-third data in Battery monitor were used in this phase. For the other two subject, we used one-sixth (4 hours out of 24 hours data) for training. We evaluated different classification algorithms against our training data. Most popular algorithms with default settings performed already well (with F-measure higher than 0.8). We use J48 for Battery monitor subject, and Multilayer Perceptron classifier for the other two subjects in our experiments for **SHAP.**

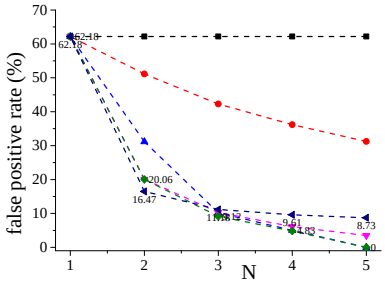
Figure 6 compares the effectiveness of different hazard suppression approaches on all three application subjects. Results of **SHAP-Max** and **SHAP**$^+$ are explicitly labeled in the figures. Generally, the false positive rate (i.e., percentage of inconsistency hazards) decreases with N's growth. We observe that **Del**, **SHAP-Min** and **SHAP-Max** are very effective in suppressing the detection of hazards when N is large (suppression rate can be more than 90%), and **SHAP** can beat **Del** even when N is very small (e.g., 2 or 3). This indicates that real inconsistencies can be reported and resolved in a more timely manner with **SHAP**. **SHAP**$^+$ is even more effective when N is very small. However, it is not so effective when N grows larger. This indicates that the most effective hazard patterns derived by **SHAP**$^+$ can be those with few (e.g., 2 or 3) context changes.

We also observe that **SHAP**$^+$ can beat **SHAP** even when N equals to 2, where **SHAP**$^+$ seemingly also considers context change pairs as hazard patterns. We note that this improvement is because more information is used in **SHAP**$^+$. When scheduling for a constraint cst, **SHAP**$^+$ considers all context changes received, while **SHAP** considers only those affecting this constraint for hazard patterns, and this makes **SHAP** less effective when it faces some situations. For example, the constraint specified in Equation (4) (denoted as cst_{12}) can be affected by changes on contexts $Light_1$ and $Light_2$. When a worker walked from Room 2 to Room 1, it would incur three changes: deleting the worker's identity from context $Light_5$ (i.e., change chg_1), adding his identity to context $Light_1$ (i.e., change chg_2) and $Light_2$ (i.e., change chg_3), respectively. These changes might be received in any order, depending on different updating rates of concerned sensors and different delays during data transmission.. If chg_2 is first received, followed by chg_1 and chg_3, a hazard would be detected if cst_{12} is evaluated after receiving chg_2. **SHAP** cannot suppress this hazard with N=2 since chg_1 does not affect cst_{12} at all, and thus chg_2 will not be scheduled to $\otimes$. Instead **SHAP**$^+$ has a chance to recognize $[chg_2, chg_1]$ as a hazard pattern, and thus suppresses this hazard.

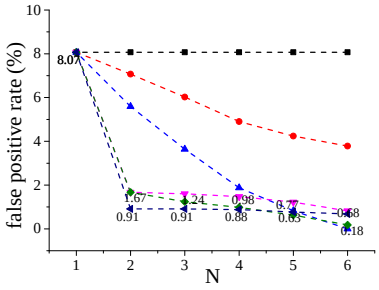
We observe that **SHAP** incurred a few missed real inconsistencies. We thus compared the false negative rate among different **SHAP** strategies for all application subjects, as illustrated in Figure 7. We explicitly labeled results of **SHAP**$^+$ in these figures. We note that the false negative rate is no more than 3% for **SHAP-Min and SHAP-Max** on all application subjects, and no more than 5% for **SHAP**$^+$ (worst case only occurs on the smart exhibition application, and the false negative rates on the other two subjects are both



(a) Smart exhibition application



(b) Smart light application

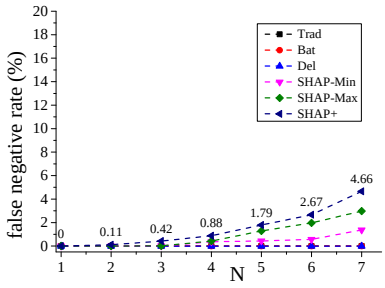


(c) Battery monitor application

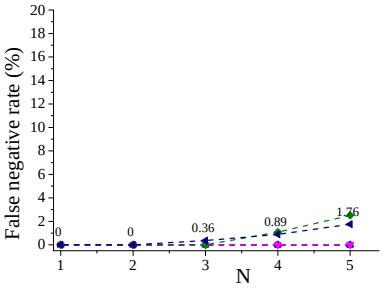
Figure 6: Hazard suppression comparisons

less than 3%), even when N is set to the maximum value (i.e., ST). When N is relatively small, the false negative rate is almost negligible (less than 1%), and thus **SHAP** is effective in both suppressing the detection of inconsistency hazards (much more effective than **Del**) and preserving the detection of real inconsistencies. **SHAP**$^+$ is more aggressive when recognizing hazard patterns, and thus has relatively higher false negative rates. Nevertheless, absolute values of false negative rate for different strategies are all very low.

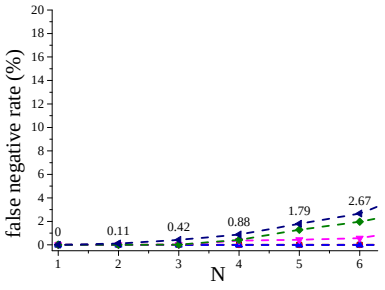
Regarding efficiency, **SHAP** and **Bat** can greatly reduce the number of conducted inconsistency detections since they selectively check contexts against constraints (**Del** cannot as it detects all inconsistencies and then waits). Figure 8 compares normalized numbers of inconsistency detections among all approaches for the three applications (**Trad**'s numbers are used as the baseline for normalization, which exactly equal to **Del**'s numbers). Results of **Bat**, **SHAP-Max** and **SHAP**$^+$ are explicitly labeled in the figures. **Bat** reduces a large portion of inconsistency detections. However, **Bat** is much less effective in hazard suppression, as discussed earlier. **SHAP** can effectively suppress the detection of hazards, and also reduce the number of conducted inconsistency detections (about 15-36% less). **SHAP**$^+$ can be even more efficient than **Bat** in reduc-



(a) Smart exhibition application



(b) Smart light application



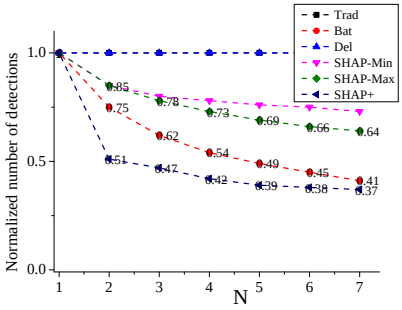
(c) Battery monitor application

Figure 7: False negative comparisons

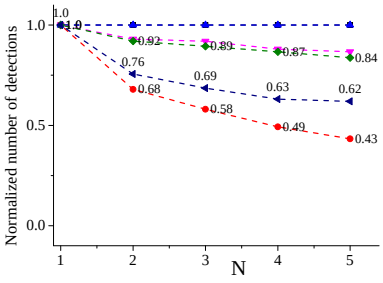
ing detections, but it comes with the cost of classifying longer context change sequences.

6.4. Discussions

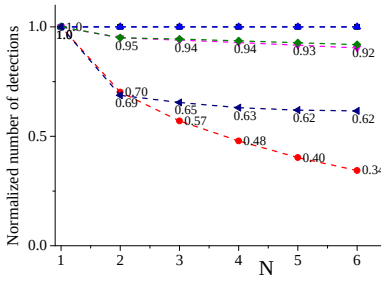
We discussed and compared both intuitive and dedicated hazard suppression approaches. We implemented them in the same framework by sharing the same context model and consistency constraint data structures. Their only difference lies in how to schedule inconsistency detection with their built-in strategies. We evaluated and compared these approaches through both real-world and simulated data, and thus the experimental results do reflect their effectiveness and differences to some extent in practice. The classification algorithms used in our experiments include Decision trees (J48), Multilayer Perceptron, etc., which are among the most popular classification algorithms. **SHAP** has two requirements when choosing classification algorithms. First, we prefer simple algorithms, especially those performing well and without dedicated parameter tuning for specific scenarios. Second, the classification process should be very efficient. Currently, classifiers are trained offline, which means long training time is tolerable. However, algorithms which are expensive in classifying unknown



(a) Smart exhibition application



(b) Smart light application



(c) Battery monitor application

Figure 8: Efficiency comparisons

instances (i.e., kNN) cannot be chosen since classifications will be conducted frequently. According to our study, most common classifiers, including Decision trees, Multilayer Perception and Tree Augmented Naive Bayesian Network (TAN) are promising for discovering hazard patterns. **SHAP** uses these classifiers implemented in **Weka** and works on automatically collected training sets.

We note that none of these techniques can beat the others in all aspects. Therefore, users should select a hazard suppression approach based on certain application requirements. For example, the **SHAP** techniques are relatively both effective and efficient. Its **SHAP-Min** strategy can protect the detection of more real inconsistencies while its **SHAP-Max** strategy can suppress the detection of more hazards. **Del** can suppress all hazards when its delay parameter is set to a sufficiently large value, but that may compromise an application's timeliness requirement. **Bat** is not very effective in suppressing the detection of inconsistency hazards, but it is very efficient by reducing a significant part of inconsistency detections.

In our experiments, only environmental contexts were considered and tested. However, our approach can also be applied to other contexts (e.g., logical contexts describing a user's mood and activity), as long as they can be modeled

in the schema described in Section 2.1. The optimal size of the dynamic buffer (N) depends on an application's requirement. Bigger the buffer is, more hazards would be suppressed, but also longer reporting real inconsistencies may be delayed. Users can choose the value of N and the strategy of inconsistency detection scheduling according to their needs.

7. Related Work

Applications in pervasive computing environments provide smart services based on context information. Many middleware infrastructures have been proposed to help applications with context acquisition, storage and management [32]. Such middleware systems include EgoSpaces [10], LIME [13], Gaia [3] and Cabot [29]. Management of environmental contexts are conducted transparently by middleware and isolated from applications. For example, ADAM [27] supports defect analysis for model-based context-aware applications. It helps find specific defects in these applications that concern predictability, stability, reachability, or other important properties. Most of these middleware systems implicitly assume that contexts are accurate, and mainly focus on assisting developers with qualified adaptation logic to provide better services. However, this assumption does not hold in reality. Thus another line of work directly focuses on the detection and resolution of context inconsistencies. For example, our previous work studied how to accelerate the inconsistency detection process by adopting incremental checking [29], concurrent checking [30] or GPU-based checking [19]. This acceleration has no essential influence on the detection scheduling and its results. Thus our research in this article is orthogonal to these pieces of work. Besides, various resolution strategies have been proposed to address detected inconsistencies [1, 28]. The problem studied in this article complements these pieces of work in that it identifies what inconsistencies should be resolved and what should not.

Sama et al. [17] analyzed common fault patterns in context-aware adaptive applications and proposed techniques to detect these faults through static analysis. They also mentioned a similar hazard problem during application adaptation [18], which echoes our studied problem in this article. However, their adaptation rules specified by propositional logic are simpler than the constraints studied in this article, which are specified by first-order logic and subject to new types of hazards.

CINA [31] addressed unstable context inconsistencies (STINs) by statically analyzing a constraint's instability conditions. STINs and hazards are similar in that they both occur during inconsistency detection, but the mechanisms to suppress them are different in these two pieces of work. First, CINA considers type information from context changes only in its derived instability conditions, while **SHAP** considers all information in context changes, including context types and concrete element values. As a result, patterns reported by **SHAP** are of a finer granularity than those by CINA. For example, CINA may consider any change pair formed by pattern $[(+, R_A, *), (-, R_A, *)]$ to be able to cause detected inconsistencies (for constraint defined in Equation (2)) unstable,

which however may not necessarily be hazards as defined in **SHAP**. Second, CINA analyzes causal relationships among contexts that cause unstable inconsistencies, while **SHAP** recognizes associations that relate to the occurrences of inconsistency hazards. As a result, **SHAP** can report patterns that are not covered by CINA, and thus suppress hazards that do not belong to unstable inconsistencies. Still, the two pieces of work has similarities, and we do have plans to further investigate their nature and combine their strengths to suppress the detection of both inconsistency hazards and unstable inconsistencies.

Hazards are also common in digital circuits. Combinational logic functions for single-input-change (SIC) operations can be realized without hazards. However, when multiple inputs are changing simultaneously (MIC), certain hazards would result [6]. Still, such hazards might be eliminated by introducing delay along certain critical paths to prevent glitches from occurring [21]. Similarly, different context sources for a context-aware application can update independently. Thus MIC is also possible for consistency constraints. It can cause inconsistency hazards to occur to applications. However, our studied problem is more complicated since the input values of digital circuits can only be zero or one, while our targeted contexts can be diverse. Besides, our consistency constraints specified in first-order logic are more complicated than combinational logic functions.

Data mining techniques are being widely used for or adapted to addressing software engineering issues. Supervised or semi-supervised learning can be used to predict whether a software model contains any bug or not [11]. Some pieces of work focus on identifying failures by discovering buggy patterns in software executions [12]. Such patterns (upon runtime execution traces) can also be used to identify root causes of failures [2, 9]. Our work in this article focuses on finding patterns in context change data to prevent potential inconsistency hazards. We made effort to label context change pairs or sequences, train classifier and decide inconsistency detection scheduling in a fully automated way. Our work requires only minimal effort from users to configure contexts' meta-information, and this makes our work applicable to a wide range of context-aware applications.

8. Conclusion

In this article, we studied the inconsistency hazard problem for context-aware applications. We analyzed the causes of such hazards and observed that they occur in certain patterns. **SHAP** techniques are proposed to schedule inconsistency detections with such patterns to suppress hazards. Our **SHAP** approach has different strategies and they have different strengths and limitations, thus catering for different suitable application scenarios.

Currently, our **SHAP** trains its classifiers to identify hazard patterns in an offline manner. At runtime, **SHAP** may encounter new hazard patterns. We are exploring for establishing a feedback mechanism to dynamically update our classification models. This would allow **SHAP** to require less training data to initialize its classifiers, but to learn hazard patterns during its inconsistency

detection. However, such runtime update will call for more computation resources, and thus need performance tuning. Besides, we also plan to investigate the relationship between the semantics of a consistency constraint and its hazard patterns. This relationship can help us better understand the nature of hazard patterns and better suppress the detection of inconsistency hazards.

Acknowledgment

This work was supported in part by National Basic Research 973 Program (Grant No. 2015CB352202), and National Natural Science Foundation (Grant Nos. 61472174, 91318301, 61321491) of China.

References

[1] Chen, C., Ye, C., Jacobsen, H.-A., March 2011. Hybrid context inconsistency resolution for context-aware services. In: Pervasive Computing and Communications (PerCom), 2011 IEEE International Conference on. pp. 10–19.

[2] Cheng, H., Lo, D., Zhou, Y., Wang, X., Yan, X., 2009. Identifying bug signatures using discriminative graph mining. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis. ISSTA '09. ACM, New York, NY, USA, pp. 141–152.
URL http://doi.acm.org/10.1145/1572272.1572290

[3] Chetan, S., Al-Muhtadi, J., Campbell, R., Mickunas, M., Jan 2005. Mobile gaia: a middleware for ad-hoc pervasive computing. In: Consumer Communications and Networking Conference, 2005. CCNC. 2005 Second IEEE. pp. 223–228.

[4] Coulouris, G. F., Dollimore, J., Kindberg, T., 2005. Distributed systems: concepts and design. pearson education.

[5] Dey, A. K., Jan. 2001. Understanding and using context. Personal Ubiquitous Comput. 5 (1), 4–7.
URL http://dx.doi.org/10.1007/s007790170019

[6] Eichelberger, E., March 1965. Hazard detection in combinational and sequential switching circuits. IBM Journal of Research and Development 9 (2), 90–99.

[7] Garcia-Molina, H., Ullman, J. D., Widom, J., 2000. Database system implementation. Vol. 654. Prentice Hall Upper Saddle River, NJ:.

[8] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I. H., Nov. 2009. The weka data mining software: An update. SIGKDD Explor. Newsl. 11 (1), 10–18.

[9] Hsu, H.-Y., Jones, J., Orso, A., Sept 2008. Rapid: Identifying bug signatures to support debugging activities. In: Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on. pp. 439–442.

[10] Julien, C., Roman, G., May 2006. Egospaces: facilitating rapid development of context-aware mobile applications. Software Engineering, IEEE Transactions on 32 (5), 281–298.

[11] Li, M., Zhang, H., Wu, R., Zhou, Z.-H., 2012. Sample-based software defect prediction with active and semi-supervised learning. Automated Software Engineering 19 (2), 201–230.
URL http://dx.doi.org/10.1007/s10515-011-0092-1

[12] Lo, D., Cheng, H., Han, J., Khoo, S.-C., Sun, C., 2009. Classification of software behaviors for failure detection: A discriminative pattern mining approach. In: Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. KDD '09. ACM, New York, NY, USA, pp. 557–566.
URL http://doi.acm.org/10.1145/1557019.1557083

[13] Murphy, A. L., Picco, G. P., Roman, G.-C., Jul. 2006. Lime: A coordination model and middleware supporting mobility of hosts and agents. ACM Trans. Softw. Eng. Methodol. 15 (3), 279–328.

[14] Probost, F., 2000. Machine learning from imbalanced data sets 101. In: Invited paper for the AAAI'2000 Workshop on Imbalanced Data Sets.

[15] Rao, J., Doraiswamy, S., Thakkar, H., Colby, L. S., 2006. A deferred cleansing method for rfid data analytics. In: Proceedings of the 32Nd International Conference on Very Large Data Bases. VLDB '06. VLDB Endowment, pp. 175–186.

[16] Raychoudhury, V., Cao, J., Kumar, M., Zhang, D., 2013. Middleware for pervasive computing: A survey. Pervasive and Mobile Computing 9 (2), 177–200, special Section: Mobile Interactions with the Real World.

[17] Sama, M., Elbaum, S., Raimondi, F., Rosenblum, D., Wang, Z., Sept 2010. Context-aware adaptive applications: Fault patterns and their automated identification. Software Engineering, IEEE Transactions on 36 (5), 644–661.

[18] Sama, M., Rosenblum, D. S., Wang, Z., Elbaum, S., 2008. Model-based fault detection in context-aware adaptive applications. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering. SIGSOFT '08/FSE-16. ACM, New York, NY, USA, pp. 261–271.

[19] Sui, J., Xu, C., Xi, W., Jiang, Y., Cao, C., Ma, X., Lu, J., Dec 2014. Gain: Gpu-based constraint checking for context consistency. In: In Proceedings

of the 21st Asia-Pacific Software Engineering Conference (APSEC 2014). Jeju, Korea, pp. 342–349.

[20] Tse, T., Yau, S. S., 2004. Testing context-sensitive middleware-based software applications. In: Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International. IEEE, pp. 458–466.

[21] Unger, S., Jun 1995. Hazards, critical races, and metastability. Computers, IEEE Transactions on 44 (6), 754–768.

[22] Wu, X., Kumar, V., Ross Quinlan, J., Ghosh, J., Yang, Q., Motoda, H., McLachlan, G., Ng, A., Liu, B., Yu, P., Zhou, Z.-H., Steinbach, M., Hand, D., Steinberg, D., 2008. Top 10 algorithms in data mining. Knowledge and Information Systems 14 (1), 1–37.
URL http://dx.doi.org/10.1007/s10115-007-0114-2

[23] Xi, W., Xu, C., Yang, W., Hong, X., 2014. How context inconsistency and its resolution impact context-aware applications. Journal of Frontiers of Computer Science and Technology 8 (4), 427.

[24] Xi, W., Xu, C., Yang, W., Yu, P., Ma, X., Lu, J., Dec 2014. Shap: Suppressing the detection of inconsistency hazards by pattern learning. In: In Proceedings of the 21st Asia-Pacific Software Engineering Conference (APSEC 2014). Jeju, Korea, pp. 414–421.

[25] Xie, H., Gu, T., Tao, X., Ye, H., Lv, J., 2014. Maloc: A practical magnetic fingerprinting approach to indoor localization using smartphones. In: Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing. UbiComp '14. ACM, New York, NY, USA, pp. 243–253.
URL http://doi.acm.org/10.1145/2632048.2632057

[26] Xie, H., Tao, X., Ye, H., Lu, J., Dec 2013. Wecare: An intelligent badge for elderly danger detection and alert. In: Ubiquitous Intelligence and Computing, 2013 IEEE 10th International Conference on and 10th International Conference on Autonomic and Trusted Computing (UIC/ATC). pp. 224–231.

[27] Xu, C., Cheung, S., Ma, X., Cao, C., Lu, J., 2012. Adam: Identifying defects in context-aware adaptation. Journal of Systems and Software 85 (12), 2812–2828.

[28] Xu, C., Cheung, S. C., Sep. 2005. Inconsistency detection and resolution for context-aware middleware support. SIGSOFT Softw. Eng. Notes 30 (5), 336–345.

[29] Xu, C., Cheung, S. C., Chan, W. K., Ye, C., Feb. 2010. Partial constraint checking for context consistency in pervasive computing. ACM Trans. Softw. Eng. Methodol. 19 (3), 9:1–9:61.

[30] Xu, C., Liu, Y., Cheung, S., Cao, C., Lu, J., Aug. 2013. Towards context consistency by concurrent checking for internetware applications. Science China Information Sciences 56 (8), Article 082105, 1–20.

[31] Xu, C., Xi, W., Cheung, S., Ma, X., Cao, C., Lu, J., 2015. Cina: Suppressing the detection of unstable context inconsistency. Software Engineering, IEEE Transactions on (forthcoming).

[32] Yang, W., Liu, Y., Xu, C., Cheung, S. C., May 2015. A survey on dependability improvement techniques for pervasive computing systems. Science China Information Sciences 58 (5), Article 052102, 1–14.