SHAP: Suppressing the Detection of Inconsistency Hazards by Pattern Learning

Wang Xi*, Chang Xu†§, Wenhua Yang*, Ping Yu†, Xiaoxing Ma†, Jian Lu†
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
Department of Computer Science and Technology, Nanjing University, Nanjing, China
*{swangex, ihope1024}@gmail.com, †{changxu, yuping,xxm,lj}@nju.edu.cn

***Abstract*—Context-aware applications rely on contexts derived from sensory data to adapt their behavior. However, contexts can be inconsistent and cause application anomaly or crash. One popular solution is to detect and resolve context inconsistencies at runtime. However, we observe that many detected inconsistencies do not indicate real context problems. Instead, they are caused by improper inconsistency detection. These inconsistencies are harmless, and their resolution is unnecessary or may even cause new problems. We name them inconsistency hazards. Inconsistency hazards should be suppressed, but their occurrences resemble normal inconsistencies. In this paper, we present a pattern-learning based approach SHAP to suppressing the detection of inconsistency hazards. Our key insight is that the detection of such hazards is subject to certain patterns of context changes. These patterns, although difficult to specify manually, can be learned effectively from historical inconsistency detection data. We evaluated our SHAP experimentally through three context-aware applications. The results reported that SHAP can automatically suppress the detection of over 90% inconsistency hazards, while preserving the detection of over 98% normal inconsistencies, with only negligible overhead.**

***Keywords*—context inconsistency; inconsistency hazards; pattern learning.**

I. Introduction

Consistency is an important property for computer systems, such as distributed systems [3] and databases [5]. Context information used by context-aware applications is also obliged to be consistent. However, these applications may run in an environment never accurate as expected due to imperfect nature of sensing devices associated with applications [7]. Context inconsistencies are thus common and may affect behavior of applications unexpectedly [15]. There is growing research on automatic detection and resolution of context inconsistencies [2][16]. Generally, one first specifies consistency constraints in some constraint language and checks an application's context information against these constraints. If any constraint is violated (i.e., evaluated to **false**), an inconsistency is said to be "*detected*". Typically, constraints should be reevaluated as soon as the application's environment changes for its adaptation timeliness. However, we observe that this common practice may be subject to numerous false alarms. We explain and illustrate this problem with a simple example below.

Consider a location-aware application deployed in a hospital scenario for assisting doctors in their daily work, e.g., notifying a doctor to take care of a nearby patient falling into an emergency situation. Various context sources can provide information for identifying such scenarios, e.g., location sensors for detecting doctors' locations and medical devices for monitoring patients' vital signs. Suppose that doctor *Lucia* leaves Room A and then enters Room B. Sensors would capture such movement events as "*Lucia disappears from Room A*" and "*Lucia appears in Room B*" to update the application's context information. However, sensors may fail to capture some events due to noises, say, missing the "*disappearing*" event. This would make the application think of *Lucia* appearing in both rooms at the same time. If we have specified a constraint like "*nobody can appear in two different rooms at the same time*", an inconsistency would result due to this constraint's violation and should be resolved before this inconsistent context is used.

However, we observe that a large number of detected inconsistencies in practice can be false alarms. In the real world, an activity can trigger a group of related changes, which are usually sensed and reported by different context sources with different update rates [11], to context information. If a constraint is evaluated when only some of these changes have taken effect while others have not yet, the constraint may behave as being violated, leading to the detection of an inconsistency. However, this inconsistency can only be 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 related contexts are ready. Such inconsistency does not require resolution, and if doing so, it may cause unexpected consequences instead. We name such false alarms **inconsistency hazards** (or hazards for short), which conceptually resemble hazards in digital circuits [14].

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

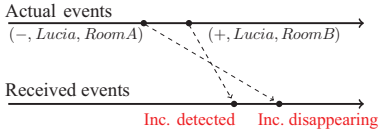


Fig. 1: Example scenario of inconsistency hazards

§Corresponding author: Chang Xu.

inconsistency detection is right scheduled between these two batches of changes (i.e., related changes are isolated), a hazard would be detected, as shown in Fig. 1.

Inconsistency hazards can be common in practice. In our investigated three context-aware applications (discussed later in evaluation), a considerable number of detected context inconsistencies are actually hazards, ranging from 8.1% to 64.9% among all. Detecting these hazards as inconsistencies wastes valuable computing resources, which should instead be used for supporting applications to better adapt to environmental changes sensitively [16]. Besides, these detected hazards would unfortunately be treated as normal inconsistencies for resolution, which may instead harm applications' functionalities (e.g., some key context information may be incorrectly updated or even deleted [18]). Therefore, hazards should be recognized or their detection should be suppressed, and this needs to be done in an automated way.

However, distinguishing hazards from normal inconsistencies is not easy as they are both caused by violation of consistency constraints. Without incorporating application-specific knowledge, there is no general way to automatically recognize hazards. In this paper, we aim to address this research issue by suppressing the detection of such hazards with a pattern-learning based approach. Our key observation is that checking of a constraint is subject to hazards only under certain patterns of context changes. These patterns may not be easily specified manually, but can be effectively learned from historical inconsistency detection data. We name our approach **SHAP**, which stands for *Suppressing Inconsistency Hazards with Pattern-Learning*. The experimental results confirmed that our approach can effectively suppress more than 90% of hazards while preserving more than 98% normal inconsistencies.

The remainder of this paper is organized as follows. Section 2 introduces background knowledge about inconsistency detection and explains our inconsistency hazard and its suppression problem. Sections 3 and 4 elaborate on our techniques for hazard suppression. Section 5 evaluates our SHAP approach experimentally. Section 6 discusses related work, and finally Section 7 concludes this paper.

II. Inconsistency Detection and Hazard

In this section we introduce some background concepts and present our inconsistency hazard problem.

A. Context and Context Changes

A *context* is a piece of environmental information used by a context-aware application. We model a *context* as a finite set of *element*s, each of which specifies a relevant part of this context. An element can have several *fields*, and each field contains a numerical or text value. For example, the current status of a doctor (e.g., **location**: *Ward3824*, **name**: *Lucia*, ...) can be an element. Then, all such elements can compose a context $DOCT$, representing all doctors in this hospital. An application can use various contexts. We use a *context pool* to contain all contexts interesting to an application. Besides $DOCT$, the pool can also contain contexts about indoor environmental information (e.g., temperature, humidity, ...) and profiles of patients.

By definition, a context can naturally support 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., **adding**, **deleting** or **updating**) 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**.

B. Inconsistency Detection

Inconsistency detection is a process of checking whether pre-specified constraints are violated or not [16]. These constraints have specified necessary conditions that must hold for a context-aware application. We express the constraints 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 the variable e can take any *element* from C as its value. Terminal *bfunc* represents any domain- or application- specific function that returns either **true** or **false** based on values of its parameters, e.g., $equals(str_1, str_2)$ or $less(v_1, v_2)$. Parameters of *bfunc* can be fields of elements or other constants. Consider a constraint "*nobody can be in two rooms (A and B) simultaneously*". It can be expressed as follows:

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

R_A and R_B are two contexts representing people currently staying in Room A and Room B, respectively. Function *equals* checks whether the two string variables contain identical contents. In common practice, whenever a related context change occurs (e.g., leaving or entering any room), all concerned constraints (including this one) are immediately checked for possible violations. We name this practice Traditional, or **Trad** for short.

C. Inconsistency Hazard

Trad can detect many context inconsistencies, which however disappear later by themselves. Such hazards do not indicate real context problems.

One reason for such hazards is the misalignment of different context sources with various sensing rates. However, even if context changes come from the same source, hazards may still occur. Let us revisit the aforementioned hospital scenario. Suppose that any ICU in the hospital having one patient inside must have at least one doctor watching on her status continuously. This requirement can be specified as a consistency constraint as follows (ICU_x represents all patients currently detected in ICU X):

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

Suppose that a doctor accompanies a patient into ICU X. The entering events are captured as two context changes, chg_1 and chg_2, which are to **update** the doctor's location to

"ICU_x" in context $DOCT$ and **add** the patient's information into context ICU_x, respectively. If the doctor pushes the patient sitting in a wheelchair into the ICU, context change chg_1 would 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 (the thing becomes clear when chg_1 is later captured and handled).

However, **Trad** does not know this inconsistency is a hazard, and will report it as a normal inconsistency immediately so it can be resolved in time. There can be many resolution strategies [2][18]. One strategy might consider context change chg_2 unreliable and choose to discard it. Thus, although the inconsistency is resolved, the application would miss the patient's information as well as failing to operate medical devices for this patient in the ICU. Or, another strategy may keep chg_2, but makes the application believe that a patient in the ICU is being unattended. As a result, warning notifications would be broadcasted to nearby doctors and such false warnings can be annoying.

Therefore, resolving hazards as normal inconsistencies is unnecessary and might even be harmful. As applications' timeliness requires, we in this paper choose to suppress the detection of such hazards. One characteristic of hazards is that they will disappear after an application handles a small set of later context changes. The size of this set is a threshold (denoted as *ST*), which can be application-specific. In this paper, we are interested in how one can predict a context inconsistency is hazard without actually detecting it when facing context changes to be handled.

III. Suppressing Inconsistency Hazards

In this section, we present several techniques to suppress the detection of hazards. The first two are intuitive, and the last one is our **SHAP** techniques based on pattern learning.

A. Intuitive Techniques

The simplest way to address hazards is a Delay-Based technique (**Del** for short). **Del** checks constraints as **Trad** does, but any detected inconsistency will not be reported or resolved until it has survived the next N (N is user-customizable and $N \leq ST$) context changes. It is clear that **Del** can preserve normal inconsistencies and remove hazards if N is set to *ST* or a sufficiently large value when *ST* is unknown. However, this is achieved at the cost of delaying reporting normal inconsistencies for up to *ST* changes. Although N can be set to a smaller value to shorten the delay, more hazards will result accordingly. This technique is suitable for scenarios where the timeliness of resolving normal inconsistencies is not emergent but no hazard can be tolerated.

The second technique is to conduct inconsistency detection every several context changes (say, N'; $N' \leq ST$). This technique (named Batch-Based, or **Bat** for short) can alleviate the hazard problem. The delay in reporting detected inconsistencies varies between 0 and N' changes ($N'/2$ on average). However, some hazards can still be detected, and its ratio cannot be controlled. **Bat**'s advantage is that the number of inconsistency detections can be greatly reduced (being about $1/N'$), and it is thus suitable for scenarios where computing resources are very limited.

TABLE I: Impact of a context change on a formula

	Add	Delete	Update
Universal	$T \rightarrow F$	$F \rightarrow T$	$F \rightarrow T$ and $T \rightarrow F$
Existential	$F \rightarrow T$	$T \rightarrow F$	$F \rightarrow T$ and $T \rightarrow F$

In the following, we look deeper to this hazard problem.

B. Impact of Context Change

We first model the impact of a context change on the truth value of a consistency constraint. When a change *chg* (**type**: *t*, **context**: *ctx*, **element**: *ele*) is handled, the truth value of a constraint *c* may be affected only if it contains a **universal** or **existential** sub-formula *f* referencing context *ctx*. The truth value of *f* may turn from **true** to **false** ($T \rightarrow F$), turn from **false** to **true** ($F \rightarrow T$), or remain unchanged. Here, we ignore the last case.

Consider a **universal** formula *f*, defined by $\forall x \in ctx(subf(\cdots))$, which is evaluated to **true** initially (i.e., satisfied). Now we have a **deleting** change *chg*. After *chg* is applied, *f* must remain to be **true**, since all the elements left in *ctx* still satisfy *subf*. However, if *f* is **false** initially and the exact only one element violating *subf* is deleted, *f*'s truth value will change from **false** to **true**. Thus the impact of a **deleting** change on a **universal** sub-formula must be $F \rightarrow T$, if any. Table I lists potential impact of a certain type of context change on the truth value of a certain type of formula. The impact of an **updating** change can be both $T \rightarrow F$ and $F \rightarrow T$, depending on the concrete element value containing in the change.

We observe that a hazard occurs always as a result of certain patterns of context changes: one with $T \rightarrow F$ impact first causes a hazard, and then a following (not necessarily immediately following) change with $F \rightarrow T$ impact makes the hazard disappear.

C. Hazard Pattern

We define a *hazard pattern* as a pair of context changes that indicates the occurrence of a hazard. Consider our earlier constraint specified by Equation (3). Hazards would occur when a patient enters an ICU, followed by a doctor who accompanies this patient. For this example, the change pair taking the form of $[(add, ICU_x, e_p), (upd, DOCT, e_d)]$ is a hazard pattern for this constraint. Here, e_p represents any element to be added to context ICU_x and e_d represents the element to be updated in $DOCT$, as explained earlier.

Other pairs, although not directly causing hazards, may indicate coming occurrences of future hazards. For example, when two patients, *Jenny* and *Penny*, walk into an ICU, followed by a doctor *Lucia*, three changes will be captured in turn: **adding** *Jenny*'s, *Penny*'s and **updating** *Lucia*'s location information. The first two **adding** changes can cause two hazards, which would be gone when the third **updating** change is handled. Therefore, we consider change pair $[(add, ICU_x, e_p), (add, ICU_x, e'_p)]$ also a hazard pattern. The underlying insight is that the two changes should be in the same group and inconsistency detection should not be scheduled between them.

We adopt classification techniques adapted from the data mining area to recognize hazard patterns and distinguish them from other change pairs. We label them as *hazard pair*s or *safe pair*s, respectively. The classification details are discussed later in Section 4.

D. SHAP Technique

Our SHAP idea uses hazard patterns to schedule inconsistency detection smartly. Consider a constraint *cst* and a context change *chg*. If *chg* does not affect c_1 (i.e., *cst* does not contain a sub-formula referencing *chg*'s concerned context), *cst* does not have to be checked. Otherwise *cst*'s truth value may change and it is subject to checking. In this case, the inconsistency detection for *chg* will be scheduled by **SHAP** automatically.

We use a dynamic buffer to support this scheduling process, as illustrated in Fig. 2. Here, *cst* is a consistency constraint under consideration, and chg_i represents the *i*th received context change. The size of this buffer may affect the effectiveness of inconsistency detection, and is thus user-customizable according to application requirements. 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**. **SHAP-min** tries to minimize loss of detected normal inconsistencies, while **SHAP-max** tries to maximize suppression of detected hazards. Algorithm 1 shows how **SHAP-min** makes scheduling decisions when a new context change chg_i is received. The constraints used by the application are considered one by one. If chg_i does not affect a constraint *cst*, there is no need to schedule inconsistency detection for this change (thus set as $\otimes$) (Lines 2-4). If *cst* is affected, **SHAP-min** will try to find out whether chg_i is prone to hazards, and make scheduling decisions accordingly (Lines 5-10). The detection can be scheduled immediately if chg_i is unlikely to form any hazard pair with other changes (i.e., decision set as $\odot$). Otherwise the decision cannot be made immediately (thus set as $\ominus$). After that, chg_i is used to update scheduling decisions for earlier changes if they have not been decided yet (Lines 11-15). Finally, scheduling decision for the earliest change in the buffer will be set as $\odot$, if it remains $\ominus$ (Lines 16-18), since it has reached the lower boundary of the buffer.

The *hazardPairProb* and *isHazardPair* functions used in Line 5 and 12 take two context changes as input and return as output a classification result. *hazardPairProb* will output a probability of these two changes being a hazard pair, while *isHazardPair* simply returns **true** or **false**. The second input parameter can be set to wildcard character "?" to explore whether there exists any change, which can form a hazard pair with the first change, as used in Line 5.

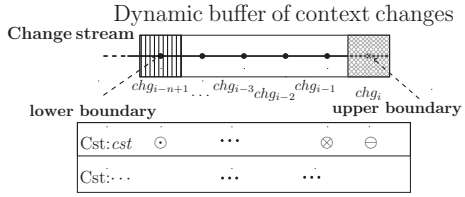


Fig. 2: Dynamic buffer for scheduling inconsistency detection

Algorithm 1 SHAP-min scheduling strategy

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

```
1:  for cst in Csts do
2:    if !cst.affectBy(chg_i) then
3:      sch(cst, chg_i) := ⊗;
4:    else
5:      if cst.hazardPairProb(chg_i, ?) < 0.01 then
6:        sch(cst, chg_i) := ⊙
7:      else
8:        sch(cst, chg_i) := ⊖
9:      end if
10:   end if
11:   for chg in buf do
12:     if sch(cst, chg_i) = ⊖ ∧ cst.isHazardPair(chg, chg_i)
        then
13:       sch(cst, chg_i) := ⊗
14:     end if
15:   end for
16:   if buf.size() ≥ N∧ sch(cst, buf.getTop()) = ⊖ then
17:     sch(cst, cst.getTop()) := ⊙
18:   end if
19: end for
```

SHAP-min can suppress the detection of most hazards, but still some may escape. Consider a change pair $[chg_i, chg_j]$ that is recognized as a hazard pair. 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 pair with it. Then inconsistency detection will still be conducted when chg_k is received (before chg_j). As a result, hazards may occur due to this scheduling (essentially caused by chg_i). We

Algorithm 2 SHAP inconsistency detection

Input: *Csts* (all constraints), *sch* (dynamic buffer for detection scheduling), *buf* (all buffered context changes); chg_i (new context change); *N* (the size of the dynamic buffer); *contextPool* (all recent contexts).
Output: *incs* (reported context inconsistencies).

```
1:  updateScheduling(chg_i);
2:  buf.add(chg_i);
3:  if buf.size()≥N then
4:    chg' := buf.getTop();
5:    buf.removeTop();
6:    ctx := contextPool.get(chg'.context);
7:    ctx.apply(chg');
8:    for cst in Csts do
9:      if sch(cst, chg) = ⊙ then
10:       check cst and report detected inconsistencies
11:     end if
12:   end for
13: end if
```

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 pair, the scheduling decision of inconsistency detection for every change between this pair (including chg_i but excluding chg_j) will be set to $\otimes$. This strategy can suppress the detection of more hazards, but might miss the detection of some normal inconsistencies. We omit this strategy's algorithm due to its similarity to **SHAP-min** and space limit.

We give the whole **SHAP** inconsistency detection process by Algorithm 2. When a new context change is received, it is first used to update detection decisions for all changes. The *updateScheduling* function in line 1 can be **SHAP-min** or **SHAP-max**, as discussed earlier. After that inconsistency detection will be conducted if necessary according to updated scheduling decisions.

IV. Learning Hazard Patterns

In this section, we introduce how to learn hazard patterns. We use **Weka** [6] to train classifiers from historical inconsistency detection data. We first represent context change pairs in a unified format readable to **Weka**. These change pairs will then be labeled as *safe pair*s or *hazard pair*s in an automated way. After that, we train classifiers based on these labeled data.

A. Representing Context Change Pairs

The input of **Weka** should be a set of **instance**s with the same features. In our problem, these features include a context change's *type*, concerned *context* and concrete *element* value. Unfortunately, different contexts may have different structures (e.g., two elements from two different contexts can have different sets of fields). We have to make all contexts share the same structure for classifier training. Therefore, we require that each context should be associated with a piece of meta-information, as illustrated in Fig. 3. We can then use a database **join** operation to merge contexts' meta-information. Accordingly, a constraint's meta-information would be the **join** result of all its referenced contexts' meta-information.

For example, suppose that context ICU_x contains two fields: **name** and **isPatient**, and context $DOCT$ contains three fields as illustrated in Fig. 3. Then the constraint given earlier by Equation (3) concerns four fields in its meta-information, i.e., **name**, **location**, **age** and **isPatient**. Consider a context change pair: [(*add*, ICU_x, (**name**: *Jenny*, **isPatient**: **true**)), (*upd*, $DOCT$, (**name**: *Lucia*, **location**: ICU_x, **age**: 26))], it will be converted into an instance as follows:

$$(add, ICU_x, Jenny, true, ?, ?, upd, DOCT, Lucia, ?, ICU_x, 26)$$

, where "?" means no value (e.g. the first change (*add* to ICU_x) does not contain values concerning **location** and **age** fields, while the second change (*update* $DOCT$) does not contain value concerning **isPatient**). This instance contains a total of 12 features, six for each change. The first two features of each change represent its *type* and *context*, and the remaining four correspond to all non-equivalent fields in these contexts.

```
<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>
```

Fig. 3: Context meta-information

B. Labeling Change Pairs

The second step is to label collected context change pairs for classifier training purposes. As illustrated in Fig. 4, we keep track of all detected inconsistencies for a 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. 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 still persists.

If an earlier inconsistency is detected after applying chg_j ($i - st + 1 \leq j < i$) and disappears after applying chg_i, this inconsistency is a hazard (e.g., inc_y, where $j = i - 2$). Accordingly, the change pairs between chg_j and chg_i will be labeled as *hazard pair*s (e.g., $[chg_{i-2}, chg_i]$, $[chg_{i-1}, chg_i]$ and $[chg_{i-2}, chg_{i-1}]$). 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 pair, then it forms a *safe pair* with each of other changes in the buffer. In case that a change pair is labeled to both *hazard pair* and *safe pair* (i.e., instances with exactly the same features are labeled differently), *hazard pair* will be selected for resolving the tie for safety.

We detect inconsistencies with **Trad** and label context changes to collect the training sets. In this process, input context changes can be either real changes or simulated changes, as long as the simulated changes satisfy the statistical distribution. Generally, with more training data, more accurate classifiers can be trained. Ten-fold cross-validation [6] is used to measure the accuracy of the trained classifier. We stop collecting new change pairs when the accuracy is good enough (F-measure [6] greater than 0.9).

C. Training Classifiers

We use **Weka** [6] to train our classifiers for predicting whether a change pair is a hazard pair or not. **Weka** is a collection of tools for data mining tasks. All **Weka**'s classifiers

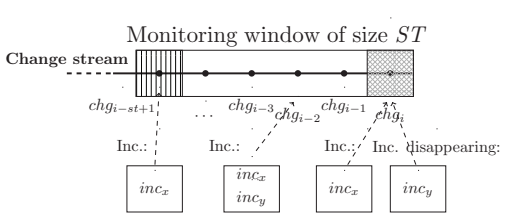


Fig. 4: Labeling context change pairs

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 of a given new instance.

The *hazardPairProb* function used in **SHAP-min** converts a pair of context changes into an instance and feeds it to *distributionForInstance* in **Weka**. Its returned value *dist* is then used to calculate whether this change pair is a *hazard pair* and with what probability. Another *isHazardPair* function used in **SHAP-min** would return **true** if this calculated probability is greater than 0.5, and **false** otherwise.

V. Evaluation

We conducted experiments to evaluate our techniques. **Trad** is used as a baseline to disclose all normal 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 7. The **independent variables** of our experiments include inconsistency detection techniques (**Trad**, **Del**, **Bat**, **SHAP-min** and **SHAP-max**), and parameter N, which decides the size of the dynamic buffer in **SHAP** techniques. 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 normal inconsistencies against all normal inconsistencies). *Efficiency* is measured indirectly by comparing *total number of constraint evaluations conducted in inconsistency detection*.

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

1) Smart exhibition application: Smart exhibition supports context-aware exhibition activities and is deployed on our campus with real hardware. It manages three exhibition rooms and a connecting corridor, as illustrated in Fig. 5, in our department building. Each room door is installed with an RFID reader to detect visitor entering or leaving events. The application uses such visitor location information to customize exhibition contents according to each visitor's interest and needs. Five constraints are deployed for this application. Three of them are designed to ensure location consistency (Type 1, similar to Equation (2)) and the other two are for ensuring the visitors in two Special Rooms being accompanied by guides (Type 2, similar to Equation (3)).

2) Smart light application: Smart light [13] can dynamically adjust the luminance of lights in a workplace according to locations of workers for saving energy, at the same time meeting minimal illumination requirements. For experimental purposes, we simulated a scenario for testing this smart light application as illustrated in Fig. 6. This scenario's physical layout naturally enforces some constraints. For example, a worker covered by Light_3 must also be covered by Light_5, but not by Light_4.

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. It runs with data collected from the UbiComp 2014 programming competition [1], which provides a massive, global dataset of real-world smartphone usage. The application uses a set of constraints to detect anomalies in such usage data, e.g., if a smartphone's charger is connected, its battery must be in a charging state.

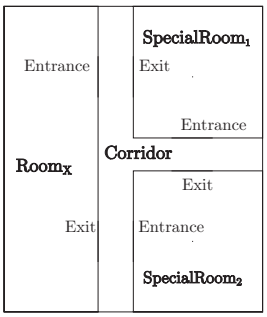


Fig. 5: Smart exhibition app.

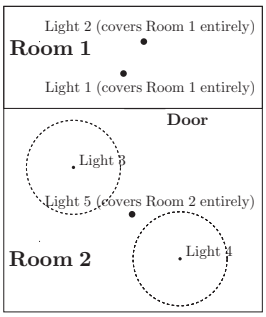


Fig. 6: Smart light app.

A. Detected Hazards

We ran Smart exhibition and Smart light for 24 hours each. Their collected context changes were 19,431 and 64,975 per hour, respectively. This corresponds to a group of 20 visitors accompanied by 3 guides and a workspace containing 10 workers, respectively, which can be considered as typical scenarios. For the Battery monitor application, its context changes were from the UbiComp 2014 data, which are 1,912,502 in total. We detected these applications for context inconsistencies with **Trad** (i.e., detecting inconsistencies eagerly). Table II lists the number of detected inconsistencies and their hazard rate for each application. We observe that the hazard rate varies from 8.1% to 64.9%, which is significant. Even for the same application (e.g., Smart exhibition), its hazard rate can be different for different types of constraints (e.g., 18.9% for Type 1 constraints and 37.8% for Type 2 constraints). This imposes challenges to techniques that should suppress the detection of hazards for various constraints.

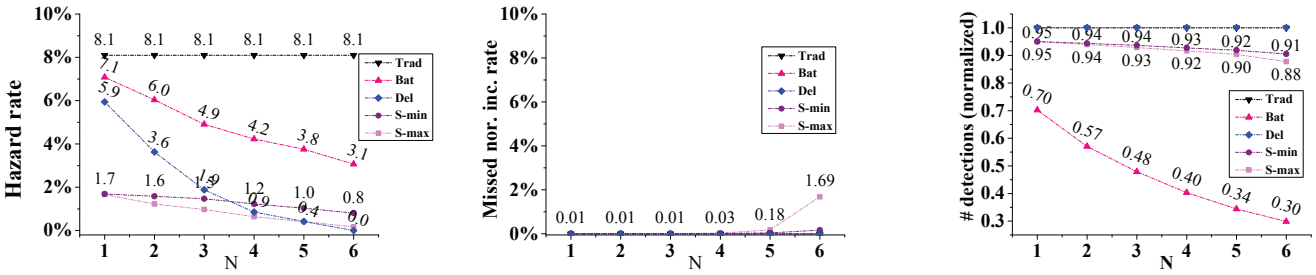
Table II also lists *ST* values, which were set according to application requirements. For example, in the Smart exhibition application, its visitors usually gathered into small groups of 5 or 6 people each, and therefore its *ST* value (indicating how long a hazard can survive) was set to 7.

B. Hazard Suppression

Since detected hazards are so common, we then evaluated whether our proposed techniques can effectively suppress the detection of these hazards. Due to page limit, we present experimental results for the Battery monitor application mostly.

TABLE II: Detected hazards

Application	*ST*	# Constraint	All incs.	Hazard rate
Smart exhibition	7	**Type 1** (3)	69,831	18.9%
		Type 2 (2)	105,554	37.8%
		Total (5)	175,385	30.3%
Smart light	5	**Total** (6)	7,955	64.9%
Battery monitor	6	**Total** (4)	58,586	8.1%



(a) False positive (hazard) rate

(b) False negative rate

(c) # conducted detections

Fig. 7: Effectiveness and side effect comparisons for different inconsistency detection techniques (on Battery monitor)

The other two applications have data with similar trends but simply different values, and we present their partial results when necessary. Fig. 7(a) compares detected hazards (false positives) among different inconsistency detection techniques for the Battery monitor application. When parameter N grows from 1 to 6, detected hazards all decrease (except **Trad**, which represents all hazards). We observe that all our techniques are effective in suppressing the detection of hazards, with an average suppression rate ranging from 63.0% to 100% when N is 6. Among them, **Del** and **SHAP** techniques (S-min and S-max in figures) worked best (more than 90%). Fig. 7(b) compares missed normal inconsistencies (false negatives) among different techniques for the Battery monitor application. We observe that all techniques are effective in preserving the detection of normal inconsistencies. Only SHAP-max incurred a little bit relatively more missed normal inconsistencies. Even so, the rate is no more than 1.7% (i.e., protecting more than 98% of normal inconsistencies) and its averaged missing rate is only 0.5%, which is not significant.

From the comparison, **SHAP** can suppress the detection of most hazards even when parameter N is set to a small value (meaning smaller delay). For example, from Fig. 7(a), **Del** has to delay at least three changes to suppress about 75% hazards, while **SHAP** requires only one to achieve the same effect. Besides, **Bat** can never beat **SHAP** within all N values.

Regarding efficiency, **SHAP** and **Bat** can reduce the number of inconsistency detections since they selectively check contexts against constraints (**Del** cannot as it detects all inconsistencies and then waits). Fig. 7(c) compares normalized numbers of detections among all techniques for the Batter monitor application (**Trad**'s numbers are used as the baseline for normalization, which exactly equal to **Del**'s numbers). **Bat** reduces inconsistency detection numbers the most (about three times compared to other techniques). However, **Bat** is much less effective in hazard suppression, as shown in Fig. 7(a). **SHAP** can effectively suppress the detection of hazards, and also reduce inconsistency detection numbers (about 10% less). In application scenarios where hazards are a lot (e.g., Smart exhibition and Smart light applications), **SHAP** can thus work both effectively and efficiently. Fig. 8 compares the efficiency in terms of normalized inconsistency detection numbers for the three applications (parameter N set to 5). We observe that SHAP reduced 10-40% detection numbers, which can save much computation effort for using by applications themselves.

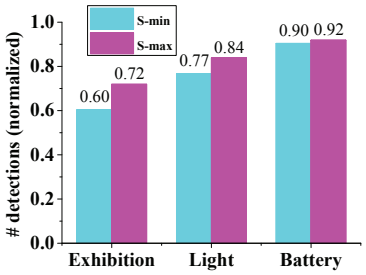


Fig. 8: # conducted inconsistency detection ($N = 5$)

C. Discussions

We implemented these techniques in the same way by sharing the same data structures for contexts and constraints. Their only difference is how to decide the scheduling of inconsistency detection. We evaluated and compared these techniques through both real-world and simulated application scenarios, and thus the experimental results do reflect their effectiveness and difference to some extent in practice.

None of these techniques can beat all others in all aspects. We argue that users should select a hazard suppression technique based on certain application requirements. For example, **SHAP** techniques are relatively both effective and efficient, in which **SHAP-min** protects more normal inconsistencies while **SHAP-max** suppresses more hazards. **Del** can suppress all hazards when its delay is set to a sufficiently large value, but that may compromise an application's timeliness requirement. **Bat** is not very effective in suppressing hazards, but it is very efficient by reducing significant inconsistency detections.

VI. Related Work

Context-aware applications are growing fast. Many middleware infrastructures have been proposed to support the development of such applications, such as EgoSpaces [8], LIME [10] and Cabot [16]. Typically, management of environmental contexts is isolated from applications and transparently conducted by such middleware infrastructures. ADAM [17] further extends Cabot by supporting 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.

The above work mainly focuses on assisting developers with qualified adaptation logics for context-aware applications. Another line of work directly focuses on validating contexts and identifying problems in contexts effectively and efficiently, with the perspective that problematic contexts can also lead applications to behave abnormally. Our previous work studied how to efficiently detect context inconsistencies [16] and resolve detected inconsistencies automatically [18]. Chen et al. also studied how to support hybrid inconsistency resolution at the service level [2]. The problem studied in this paper complement these pieces of work in that it identifies what inconsistencies should be resolved and what should not.

Sama et al. [11] analyzed common fault patterns in context-aware adaptive applications and proposed techniques to detect these faults through static analysis. They also mentioned the hazard problem in application adaptations [12], which echoes our studied problem in this paper. However, their adaptation rules specified by propositional logic are simpler than the constraints studied in this paper, which are specified by first-order logic and subject to new types of hazards. Besides, their work might mistakenly report hazards as normal inconsistencies since static analysis naturally contains false positives.

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 [4]. Still, such hazards might be eliminated by introducing delay along certain critical paths to prevent glitches from occurring [14]. Similarly, different context sources for a context-aware application can update independently. Thus MIC is typical for consistency constraints. It can cause inconsistency hazards occurring to applications. However, our studied problem is more complicated since input values of digital circuits can only be zero or one, while our targeted contexts are diverse. Besides, our constraints specified in first-order logic are more complicated than combinational logic functions.

Pattern-based learning has been adopted to locate bugs in software by discovering buggy patterns in software execution [9]. Our work focuses on finding patterns in context change data to prevent potential hazards. We made effort to label context change pairs, 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.

VII. Conclusion

In this paper, we studied the hazard problem in context inconsistency detection for context-aware applications. We proposed several techniques to suppress the detection of such hazards. These techniques have different strengths and limitations, and thus have different suitable application scenarios. Although our proposed techniques seem to apply to different applications and constraints according experimental evaluation, we still need to validate their effectiveness in a wider range of application scenarios. Besides, we plan to further investigate how to better balance the loss of normal inconsistencies and suppression of hazards, as well as minimizing the delay caused by change pair prediction. We are working along this line.

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, 61361120097) of China.

References

[1] "UbiComp/ISWC 2014 Programming Competition," http://ubicomp.org/ubicomp2014/calls/competition.php.

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

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

[4] E. Eichelberger, "Hazard detection in combinational and sequential switching circuits," *IBM Journal of Research and Development*, vol. 9, no. 2, pp. 90–99, March 1965.

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

[6] M. Hall, E. Frank *et al.*, "The weka data mining software: An update," *SIGKDD Explor. Newsl.*, vol. 11, no. 1, pp. 10–18, Nov. 2009.

[7] K. Henricksen, J. Indulska, and A. Rakotonirainy, "Modeling context information in pervasive computing systems," in *Proceedings of the First International Conference on Pervasive Computing*, ser. Pervasive '02. London, UK, UK: Springer-Verlag, 2002, pp. 167–180.

[8] C. Julien and G. Roman, "Egospaces: facilitating rapid development of context-aware mobile applications," *Software Engineering, IEEE Transactions on*, vol. 32, no. 5, pp. 281–298, May 2006.

[9] D. Lo, H. Cheng, J. Han, S.-C. Khoo, and C. Sun, "Classification of software behaviors for failure detection: A discriminative pattern mining approach," ser. KDD '09. New York, NY, USA: ACM, 2009, pp. 557–566.

[10] A. L. Murphy, G. P. Picco, and G.-C. Roman, "Lime: A coordination model and middleware supporting mobility of hosts and agents," *ACM Trans. Softw. Eng. Methodol.*, vol. 15, no. 3, pp. 279–328, Jul. 2006.

[11] M. Sama, S. Elbaum, F. Raimondi, D. Rosenblum, and Z. Wang, "Context-aware adaptive applications: Fault patterns and their automated identification," *Software Engineering, IEEE Transactions on*, vol. 36, no. 5, pp. 644–661, Sept 2010.

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

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

[14] S. Unger, "Hazards, critical races, and metastability," *Computers, IEEE Transactions on*, vol. 44, no. 6, pp. 754–768, Jun 1995.

[15] W. Xi, C. Xu, W. Yang, and X. Hong, "How context inconsistency and its resolution impact context-aware applications," *Journal of Frontiers of Computer Science and Technology*, vol. 8, no. 4, p. 427, 2014.

[16] C. Xu, S. C. Cheung, W. K. Chan, and C. Ye, "Partial constraint checking for context consistency in pervasive computing," *ACM Trans. Softw. Eng. Methodol.*, vol. 19, no. 3, pp. 9:1–9:61, Feb. 2010.

[17] C. Xu, S. Cheung, X. Ma, C. Cao, and J. Lu, "Adam: Identifying defects in context-aware adaptation," *Journal of Systems and Software*, vol. 85, no. 12, pp. 2812–2828, 2012.

[18] C. Xu, X. Ma, C. Cao, and J. Lu, "Minimizing the side effect of context inconsistency resolution for ubiquitous computing," in *Proceedings of the 8th ICST International Conference on Mobile and Ubiquitous Systems, LNICST104*. Copenhagen, Denmark, Dec 2011, 2012, pp. 285–297.