CINA: Suppressing the Detection of Unstable Context Inconsistency
Chang Xu, Wang Xi, S.C. Cheung, Xiaoxing Ma, Chun Cao and Jian Lu

Abstract—Context-aware applications adapt their behavior based on contexts. Contexts can, however, be incorrect. A popular means to build dependable applications is to augment them with a set of constraints to govern the consistency of context values. These constraints are evaluated upon context changes to detect inconsistencies so that they can be timely handled. However, we observe that many context inconsistencies are unstable. They vanish by themselves and do not require handling. Such inconsistencies are detected due to misaligned sensor sampling or improper inconsistency detection scheduling. We call them unstable context inconsistencies (or STINs). STINs should be avoided to prevent unnecessary inconsistency handling and unstable behavioral adaptation to applications. In this article, we study STINs systematically, from examples to theoretical analysis, and present algorithms to suppress their detection. Our key insight is that only certain patterns of context changes can make a consistency constraint subject to the detection of STINs. We derive such patterns and proactively use them to suppress the detection of STINs. We implemented our idea and applied it to real-world applications. Experimental results confirmed its effectiveness in suppressing the detection of numerous STINs with negligible overhead, while preserving the detection of stable context inconsistencies that require inconsistency handling.

Index Terms—Constraint, context inconsistency, impact propagation, instability analysis, pervasive computing.

1 INTRODUCTION

Context-aware applications are gaining popularity with proliferated uses of mobile and sensing devices. They adapt their behavior to deliver smart services based on contexts about their environments. For example, a location-aware application can select the nearest printer to print user documents by considering user location context. However, contexts are naturally subject to sensing noises (e.g., sensory data can be wrong, missing or redundant), and this gives rise to the context inconsistency problem [58], [61] (e.g., a user’s location is wrongly calculated, has multiple incompatible values, or contradicts with history location or other contexts). Context inconsistency occurs when contexts become inaccurate, incomplete, or conflicting with each other. Inconsistent contexts can lead to application misbehavior, e.g., user documents being printed at an unexpected location. Thus they should be timely detected and handled for the provision of dependable context-aware applications. Context inconsistency can be detected by checking contexts against a set of rules, known as consistency constraints [38], [45], [61] that assert the legitimacy of context values. If any constraint is violated, context inconsistency is said “detected”. Follow-up actions can then be taken to handle the problem. If an application has not yet used inconsistent contexts, parts of the contexts can be used after inconsistency resolution [59], [60]. If the application has already used inconsistent contexts, exception handling mechanism can be invoked to compensate the effects caused by such contexts [11], [34]. Thus managing context consistency helps towards the quality of contexts and also of the applications that run with these contexts.

During our study of context inconsistency, we observed that many detected context inconsistencies are unstable. Such inconsistencies can vanish by themselves and their handling is unnecessary. They are detected not because the concerned contexts are inaccurate, incomplete, or conflicting with each other. Instead, they are detected due to misaligned sensor sampling or improper inconsistency detection scheduling. Misaligned sensor sampling occurs when sensory data updates are disordered due to asynchronous sensor sampling and they are not synchronized before inconsistency detection. Improper inconsistency detection scheduling occurs when contexts related in a constraint have only been partially updated before inconsistency detection. In other words, unstable inconsistencies occur because their detection is performed at a time when contexts are not ready. If the detection is performed at a time when contexts are ready, these inconsistencies would not have been detected (we give examples later).

Detecting and handling unstable inconsistencies can waste valuable computational resources and may even cause undesirable consequences. For example, if an aggressive handling strategy [7], [13], [60] discards a context that is misjudged as inconsistent, the concerned application may miss key behavioral adaptation that can otherwise be triggered by this context. If the application has already used this context and conducted corresponding adaptation, the invoked exception handling for this inconsistency has to compensate the effects of this adapta-
tion. This would cause the application’s behavioral instability [46], [62], as well as the value loss from expensive compensation [65]. Later, this handling effort turns out to be meaningless as this inconsistency does not have to be handled. However, this application has already behaved unstably or even caused value loss, defying the purpose of context inconsistency detection and handling.

1.1 Examples

Unstable inconsistencies and other stable ones can coexist naturally in pervasive computing, and unstable inconsistencies can be quite common. For example, our experiments with an exhibition application revealed that 25.8–72.3% of its detected context inconsistencies were unstable. This rate increased to 30.7–80.0% for another context-aware application Smart Light [50]. A large-scale study conducted using 1.5 million real taxi data in a Smart City application [61] even found over 90% unstable context inconsistencies (we give details later in evaluation). To understand why unstable inconsistencies are so common, let us review how software engineers guarantee the consistency of software artifacts in development. Software engineers usually perform inconsistency detection before and after any changes to software artifacts [45]. As such, inconsistencies introduced by the changes can be disclosed by comparing the two detection results. However, in pervasive computing contexts are subject to rapid and continual changes [58], [61]. How to determine an appropriate schedule for inconsistency detection (i.e., deciding when to perform inconsistency detection) and handling is unclear. For example, is it appropriate to perform inconsistency detection upon each collected context change, or only after collecting a group of context changes? Which schedule(s) would suffer the detection of unstable inconsistencies? If the latter schedule helps reduce unstable inconsistencies, how should context changes be grouped for inconsistency detection? Currently, there are no clear answers to such questions.

We discuss some real-life examples of unstable inconsistency. They are from an exhibition application deployed on our campus, which aims to deliver context-aware services to exhibition visitors. Exhibitions are held in a building consisting of several halls. Each visitor wears a radio frequency identification (RFID) tag so that the visitor’s location can be continually tracked. The location information allows each hall’s exhibition contents, such as narration, to be dynamically customized to suit visitors’ interests. Most halls allow free visits, but one hall exhibits a special collection. Visitors entering this special hall must be accompanied by a guide. We specify this requirement as a consistency constraint: If any visitor appears in the special exhibition hall, at least one guide must be present. It can be formulated as: \( (\exists v \in V) \{ true \} \) implies \( (\exists g \in G \{ g.loc == \text{"hall s"} \}) \). Here, \( V \) is a context representing the set of visitors currently in the special hall \( s \), and \( G \) is a context representing the set of all guides in the whole exhibition area. Although this constraint captures the consistency requirement, problems can arise in its deployment. Suppose that three visitors accompanied by a guide enter the special hall in turn. This causes four context changes that concern the presence of people in this hall. The context change for the guide’s presence is, however, not necessarily the first one. If inconsistency detection is performed upon each context change, it would report context inconsistency once a visitor is found in this hall without the presence of the guide. We consider this inconsistency unstable because it is detected at a time between the arrival of this visitor and that of the guide. It lasts until the application detects the guide’s presence, and then the inconsistency vanishes. The handling of this inconsistency is unnecessary because it does not indicate any context problem. It is detected simply because the inconsistency detection is performed at inappropriate time. To see this, we consider another schedule. If one performs inconsistency detection only after the guide has entered the hall, this inconsistency would not have been detected. This is a desirable detection schedule without reporting such unstable inconsistency. Fig. 1 illustrates this scenario and compares the two schedules. This is an example of improper inconsistency detection scheduling (scheduled when the visitor context is updated but the guide context has not yet). Note that solving this problem is not that seemingly simple. For example, why does updating the guide context seem so special as compared to updating the visitor context, such that inconsistency detection is better scheduled for the former only? Unfortunately, this actually does not hold generally. Our later analysis will show that inconsistency detection indeed should not be scheduled for the \( v_3 \) change but should for the \( v_2 \) change. We will explain it later.

We then consider another constraint for ensuring the consistency of visitor locations: No visitor can appear in two halls at the same time. It can be formulated as: \( \forall v_i \in V_h \{ \text{not } (\exists v_j \in V_y \{ v_i.id == v_j.id \}) \} \), where \( V_h \) and \( V_y \) are two contexts representing the sets of visitors currently in halls \( x \) and \( y \), respectively. Similar problems can arise when a visitor (say \( v_0 \)) leaves hall \( x \) and then enters hall \( y \). This triggers two context changes: “deleting \( v_0 \) from \( V_x \)” and “adding \( v_0 \) to \( V_y \)”. If context changes are asynchronously collected
[46] by RFID readers deployed in different halls with different sampling cycles, “adding $v_0$ to $V_v$” can be handled before “deleting $v_0$ from $V_v$.” If inconsistency detection is performed upon each context change, it would report an unstable inconsistency when handling “adding $v_0$ to $V_v$.” This inconsistency lasts until the application handles “deleting $v_0$ from $V_v$.” The handling of this inconsistency is unnecessary because it can vanish itself. If one is smart enough to perform inconsistency detection only after both context changes have been collected, this inconsistency would not have been detected. This is a desirable detection schedule without reporting such unstable inconsistency. Fig. 2 illustrates this scenario and compares the two schedules. This is an example of misaligned sensor sampling (sensory data updates are disordered and they are not synchronized before inconsistency detection). Note that making the same sampling cycle for all RFID readers is possible but does not necessarily solve the problem. Data transmissions between distributed devices often suffer varying delay and time drift [21], [32], leading to inevitable asynchrony in sensory data collection. Besides, such sampling and delay can introduce other problems as we explain below.

1.2 Challenges

The preceding two examples discuss different consistency constraints, but their detected context inconsistencies are both unstable. We name them unstable context inconsistencies (“STIN” for short, with a plural form of “STINs”). It is desirable to distinguish STINs from other stable inconsistencies or even make STINs undetected. Unfortunately, both STINs and stable inconsistencies are detected due to violation of consistency constraints. They behave similarly. One may propose avoiding STIN occurrences by synchronizing the sampling of all sensing devices (i.e., collecting their status values in a common cycle and at the same time). However, sampling cycles of sensing devices can vary significantly [46]. For example, on the J2ME platform a GPS sensor typically refreshes every few seconds [25], and a Bluetooth sensor refreshes roughly once per minute [24]. On the Android platform, sensors have customizable sampling cycles from milliseconds to minutes, and a tricky thing is that Android might use a smaller delay than the one developers specify, leading to less controllable sampling cycles [3]. On the iOS platform, acceleration sensing is suggested at a rate of 10-20 Hz for orientation detection but 70-100 Hz for hitting or shaking detection [23], and location notifications are expected once every five minutes or over 500 meters [22]. Thus synchronizing the sampling of various sensing devices can lead to a fairly large common cycle, which may not be acceptable for applications. Besides, misaligned sensor sampling is only one reason for STIN occurrences. The preceding first example also shows that even if sensor sampling is synchronized, immediate scheduling of inconsistency detection (i.e., performing inconsistency detection upon each context change) can still detect STINs. Therefore, STINs cannot be assumed easily avoidable.

Another possible solution is to delay reporting context inconsistencies to see whether earlier detected inconsistencies can vanish themselves. If yes, they are STINs and should not be reported. However, determining a suitable length value for the delay can be difficult. To make sure that all detected context inconsistencies are not STINs, the delay has to be set to a sufficiently large value. This would reduce an application’s responsiveness to its context changes, and can be unacceptable when the application involves time-critical operations (e.g., for vehicle control and healthcare applications [19], [36]). In fact, STIN durations are not necessarily short. For example, a 10-second delay was required in our exhibition application to recognize all STINs in its field test. The delay was increased to 40 seconds for the Smart Light application (we give details later in evaluation). Such large delays caused inappropriate contents shown to uninterested visitors or wrongly controlled lights in warehouse for the two applications. Besides, even if people might tolerate such delays in certain scenarios, these STINs have to be first detected and then ignored, leading to wasted computational resources. This can cause extra problems to context-heavy scenarios, which have to exploit all computing capabilities to handle huge volumes of contexts efficiently. For example, our study of the Smart City application with millions of taxi data reveals that 31.6% stable context inconsistencies were missed due to such computational resource waste during rush hours. This caused corresponding context problems undetected and the application’s route scheduling functionality affected. Therefore, avoiding STINs by delayed reporting is unacceptable in practice, or has quite restricted usage if not infeasible.

1.3 Our Contributions

In this article, we address the STIN issue from a different perspective. We propose a novel approach to suppressing the detection of STINs (i.e., making STINs undetected). This approach can automatically decide when to and when not to perform inconsistency detection so as to suppress the detection of STINs. Meanwhile, it also preserves the detection of stable context inconsistencies (i.e., those that will not vanish when contexts become ready later). This ability is important as otherwise applications would still suffer context inconsistencies and behave abnormally. To fulfill this goal, our approach makes a key observation that upon a sequence of incoming context changes, only certain patterns of their combinations can cause the detection of STINs. Such patterns are constraint-specific and we name them instability conditions. We propose algorithms to systematically derive such instability conditions from consistency constraints, and dynamically match them against incoming context changes to decide how inconsistency detection should be scheduled. We propose a family of scheduling algorithms with different trade-offs. They are designed to suit diverse needs for different applications, from having zero tolerance to any STIN to not allowing any missing of stable context inconsistency. Besides, since this approach makes STINs undetected, it also saves computational resources that would otherwise be unnecessarily wasted.

To be clear, we note that deploying our STIN suppression algorithms requires a centralized architecture for
handling context changes. That is, a middleware infrastructure receives context changes from various sensors and detects context inconsistencies for upper-layer applications. The middleware contains a centralized context pool as explained later in Section 2, but its connected sensors and applications can be distributed. It is also possible to make the middleware itself distributed, i.e., a set of middleware hosts. Then each host needs cooperation from other hosts for forwarding context changes. In that case, our work requires additional effort for a distributed setting, e.g., extra context registration and forwarding mechanisms, as well as corresponding algorithm adjustments. Still, the middleware itself can be made transparent and seemingly centralized to its applications. We have made initial attempts to evaluate consistency constraints in a distributed way [57], but that can incur large communication overhead. Therefore, we in this article assume a centralized architecture for handling context changes and suppressing the detection of STINs, and its distributed counterpart needs further effort as our future work.

The remainder of this article is organized as follows. Section 2 presents preliminary concepts and problem formulation. Section 3 elaborates on our constraint instability analysis process. Section 4 presents our scheduling algorithms to suppress the detection of STINs based on our analyzed instability conditions. Section 5 introduces our implementation and evaluates our work experimentally with real-world applications. Finally, Section 6 discusses related work and Section 7 concludes this article.

2 PRELIMINARIES

In this section, we define some concepts to facilitate subsequent discussions. We then present the formulation of the STIN problem and an overview of our approach.

2.1 Concepts

A context is a piece of environmental information interesting to an application. We model a context by a finite set of elements, each of which specifies a relevant part of this context. For example, in our exhibition application the visitors currently in hall x can be represented by a context $V_x := \{v_1, v_2, \ldots\}$. Each element $v_i$ identifies a visitor currently staying in this hall. Each element can contain application-specific fields (e.g., id and age) for additional information. This context $V_x$ can be used by the exhibition application to count the number of visitors currently staying in hall x as well as recognizing their identities.

A context change specifies what is to be changed to a context. A context change can be an addition change (adding a new element into a context), deletion change (deleting an existing element from a context), or update change (updating the value of an existing element in a context). In our exhibition application, if a visitor $v_3$ enters hall x, it will trigger an addition change. This change can be represented as: $<\text{addition}, V_x, v_3>$. If a visitor $v_1$ leaves hall y, it will trigger a deletion change and the change can be represented as: $<\text{deletion}, V_y, v_1>$. If a guide $g_0$ walks from hall x to hall s, it will trigger an update change and the change can be represented as: $<\text{update}, G, g_0, g_0>$, where $g_0.$loc == "hall x" and $g_0.$loc == "hall s".

To facilitate our discussions, we conceptually assume the availability of a context pool to support context-aware applications. The context pool is responsible for maintaining contexts derived from environmental changes, and applications can access their interesting contexts in it. The concept of context pool (or similar ones like context space or tuple space) is supported by various existing context-aware middleware infrastructures or frameworks [9], [27], [37], [42], [58]. When context changes are collected from environments, they are applied to the contexts stored in this pool. Let the contexts in the pool be $P$ (we may also call the pool $P$ directly for convenience). Since the pool’s contained contexts vary with time, we represent them as $P_0, P_1, \ldots, P_n$, each of which corresponds to a different time point, $t_{0}, t_{1}, \ldots, t_{n}$. Suppose that at time point $t_{i+1}$, $P_i$ receives a context change $\text{chg}_{i+1}$ and evolves to $P_{i+1}$. We represent this as: $P_{i+1} := \text{apply}(P_i, \text{chg}_{i+1})$, where $i \geq 0$. In this article, we assume that each context change is collected at a unique time point.

The contexts in the pool can be checked for consistency. If any consistency constraint is violated, context inconsistency is detected. Thus consistency constraints specify necessary properties that must hold about the contexts in the pool. We express consistency constraints in a first-order logic based constraint language [61] as follows:

\[
\begin{align*}
f &:= \forall \gamma \in C [f] | \exists \gamma \in C [f] | \\
&\quad \text{(f and (f | (f or f) | (f implies (f) | not (f) | predicate(v_1, \ldots, v_0)) true | false.}
\end{align*}
\]

$C$ represents a context from the context pool, and $v$ is a variable that can take any element from C as its value. Terminal $\text{predicate}$ represents any application-specific function that returns true or false based on the values of its parameters ($v_1, \ldots, v_0$). When $\text{predicate}$ is an “equal” function, it can be written as “==” for convenience. Terminals “true” and “false” are special predicates that dictate return true and false, respectively. Our preceding two constraints have been expressed in this language:

\[
\begin{align*}
\text{shall}: & (\exists v \in V_x) \text{ implies } (\exists g \in G [g.\text{loc} == "hall s")}. \\
\text{shoc}: & \forall v \in V_x \text{ not } (\exists v \in V_x \text{ [v.id == v.\text{id}])].
\end{align*}
\]

2.2 Problem Formulation

Given a consistency constraint $s$, we use check($P, s$) to denote the checking of contexts in pool $P$ against $s$. Context inconsistency is detected if check($P, s$) returns false.

Suppose that at time point $t_i$, there is no context inconsistency detected against constraint $s$ for contexts in pool $P_i$, i.e., check($P_i, s$) == true. Let $\text{chg}_{i+1}$ and $\text{chg}_{i+2}$ be two context changes collected at the next two time points $t_{i+1}$ and $t_{i+2}$. If check($P_{i+1}, s$) == false and check($P_{i+2}, s$) == true, we say that inconsistency detection with constraint $s$ at time point $t_{i+1}$ is subject to STIN. Here, $P_{i+1} := \text{apply}(P_i, \text{chg}_{i+1})$ and $P_{i+2} := \text{apply}(P_{i+1}, \text{chg}_{i+2})$. It suggests that scheduling inconsistency detection at time point $t_{i+1}$ can report an unstable context inconsistency (i.e., STIN), which will vanish at time point $t_{i+2}$. This STIN lasts from $t_{i+1}$ to $t_{i+2}$, i.e., a window of size one (measured by the difference in subscript). We name such STIN vitality-1 STIN. A STIN can also last longer than a window of size one, say, from time
point \( t_{i+1} \) to \( t_{i+k} \) \((k \geq 2)\), i.e., \( \text{check}(P_{i+k}, s) == \text{false} \) (0 \( \leq i \leq k - 1 \)) and \( \text{check}(P_{i+k}, s) == \text{true} \). We name such STIN \textit{vitality}-\( n \) STIN. In this article, we aim for smart scheduling of inconsistency detection such that the detection of \textit{vitality}-\( n \) STINs can be suppressed automatically \((n \geq 1)\).

### 2.3 Approach Overview

Our approach exploits the concept of instability condition to suppress the detection of STINs. Instability conditions explain what combinations of incoming context changes can lead to the detection of STINs. With this information, one can decide whether or not to perform inconsistency detection upon certain context changes, so as to suppress the detection of STINs.

Take our earlier Scenario 1 (Fig. 1) for example. Its first three context changes are: “\( v_1 \) enters”, “\( g \) enters” and “\( v_2 \) enters”, respectively. Our approach would derive four instability conditions for the consistency constraint \( \forall \text{context} \) used in this scenario. The first context change pair, “\( v_1 \) enters” and “\( g \) enters”, matches one of the four conditions, and therefore inconsistency detection would not be performed between them (i.e., after change “\( v_1 \) enters” is collected but before change “\( g \) enters” is collected). The next pair, “\( g \) enters” and “\( v_2 \) enters”, does not match any instability condition, and therefore inconsistency detection would be performed between them. This is exactly the desirable Schedule 2 as shown in Fig. 1.

We illustrate our approach in Fig. 3. It consists of two parts: constraint instability analysis (Section 3) and inconsistency detection scheduling (Section 4). The first part takes consistency constraints and derives their instability conditions. These conditions are then used by the second part to make scheduling decisions, i.e., deciding whether or not to perform inconsistency detection upon certain context changes. The first part works in an offline manner, and instability conditions are derived only once for each consistency constraint. The second part works in an online manner, and inconsistency detection is performed selectively in a centralized middleware infrastructure against context changes received from distributed sensing devices. We elaborate on the methodology of these two parts in the following two sections in turn.

### 3 Constraint Instability Analysis

In this section, we discuss how to derive instability conditions from consistency constraints. We name this derivation process \textit{constraint instability analysis}.

Our analysis would first construct a syntax tree for each consistency constraint, and make the nodes in the tree represent all formulae used in this constraint. Then this syntax tree is used for studying how its corresponding constraint is evaluated for its truth value. When a context change affecting this constraint occurs, some nodes in the tree would need reevaluation if they have a child node or themselves reference the changed context. We refer to this reevaluation requirement as \textit{impact} of context change as it may affect this constraint’s truth value. In this section, we first model the impact of a context change (Subsection 3.1), and then study how the impact propagates along a syntax tree as well as changing its corresponding constraint’s truth value (Subsection 3.2). Finally, we derive certain patterns of context changes as instability conditions when they have a chance of causing a constraint’s truth value to change unstably, i.e., incurring STINs (Subsection 3.3).

Before we elaborate on the whole constraint instability analysis process, we illustrate it using an example at a high level. Consider the aforementioned \( \forall \text{loc} \) constraint: \( \forall v_1 \in V_e \) [not \((\exists v_2 \in V_e \ [v_1.id == v_2.id])]\). Our analysis first constructs its syntax tree, and then examines what impact would occur if a certain context change occurs to a certain place in this constraint. This impact is named \textit{initial impact}, and after its propagation to the entire constraint, it grows into \textit{ultimate impact}. For example, if a deletion change occurs to the \( V_e \) context (e.g., a visitor leaves hall \( y \)), our analysis would identify its ultimate impact as “never changing this constraint’s truth value from true to false but other changes are possible”. Our analysis would examine the impact of all types of context change to this constraint exhaustively, and figure out from all combinations certain patterns of context changes that, if occur in this order, can cause the constraint’s truth value to change from true to false and then back to true. Such patterns are reported as instability conditions for guiding the scheduling of inconsistency detection to suppress the detection of STINs, as explained later in Section 4.

#### 3.1 Impact Modeling

Our constraint instability analysis does not require concrete element information. Therefore, we represent a context change by a short form of \(<\text{type}, \text{place}>\). \text{Type} specifies whether it is an addition, deletion or update change. \text{Place} specifies the context to which this change is made. By this form, our preceding three example context changes can be expressed as: \(<\text{addition}, V_e >\), \(<\text{deletion}, V_e >\) and \(<\text{update}, G >\). They represent that an element is added into context \( V_e \), deleted from context \( V_e \), and updated in context \( G \), respectively.

A context change \textit{affects} a consistency constraint if any formula in this constraint references the context concerned by this change. Since only universal \((\forall \forall e \in C)\) and existential \((\exists y e \in C)\) quantifiers can reference contexts in the constraint language (Subsection 2.1), the formulae that are
directly affected by a context change must be those involving such quantifiers. We name such formulae initial-impact formulae. They denote the formulae where the impact of a context change initially occurs. Note that although a context change concerns one context only by definition, the context can be referenced multiple times in a constraint. Thus a context change may correspond to more than one initial-impact formula in a constraint. We give several examples. Consider the sloc constraint: “$$\forall v_1 \in V_x \ [\text{not} (\exists v_2 \in V_y \ [v_1.id == v_2.id])]$$”. Given an addition change <addition, $$V_\triangleright$$>, it corresponds to one initial-impact formula in this constraint: “$$\forall v_1 \in V_x \ [\text{not} (\exists v_2 \in V_y \ [v_1.id == v_2.id])]$$”. Given a deletion change <deletion, $$V_\triangleright$$>, it corresponds to another initial-impact formula: “$$\exists v_2 \in V_y \ [\text{true}]$$”. Then consider another constraint shall: “$$\exists v_2 \in V_y \ [\text{true}]$$”. Given an update change <update, $$G$$>, it also corresponds to one initial-impact formula in this constraint: “$$\exists g \in G \ [g \text{ loc } == \text{ "hall s"}]$$”.

We then discuss initial impact. Initial impact denotes possible changes to truth values of initial-impact formulæ due to a given context change. There are totally four cases as follows:

1. $$p_{\triangleright}$$: true $$\rightarrow$$ true;
2. $$p_{\triangleright}$$: true $$\rightarrow$$ false;
3. $$p_{\triangleright}$$: false $$\rightarrow$$ true;
4. $$p_{\triangleright}$$: false $$\rightarrow$$ false.

The first case means that the truth value of an initial-impact formula remains to be true (i.e., no change). The second case means that its truth value changes from true to false. Other cases can be explained similarly. All four cases are named four truth value changes. They are fundamental and complete. From them, we define initial impact matrix in Table 1, for modeling all possible and impossible truth value changes due to a context change occurring to an initial-impact formula.

We model initial impact by a tuple of $$(P_{pos}, P_{imp})$$. $$P_{pos}$$ is a set containing all possible truth value changes, and $$P_{imp}$$ contains those impossible. Let $$P_{all} = \{p_{\triangleright}, p_{\triangleright}, p_{\triangleright}, p_{\triangleright}\}$$, i.e., the universal set. We have $$P_{pos} \cup P_{imp} = P_{all}$$ and $$P_{pos} \cap P_{imp} = \emptyset$$. Table 1 gives the initial impact caused by a context change occurring to an initial-impact formula (3 x 2 = 6 combinations). For example, given an addition change

<table>
<thead>
<tr>
<th>$$(P_{pos}, P_{imp})$$</th>
<th>Addition change</th>
<th>Deletion change</th>
<th>Update change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Universal formula</td>
<td>$${p_{\triangleright}, p_{\triangleright}, p_{\triangleright}, p_{\triangleright}}$$</td>
<td>$${p_{\triangleright}, p_{\triangleright}, p_{\triangleright}, p_{\triangleright}}$$</td>
<td>$${p_{\triangleright}, p_{\triangleright}, p_{\triangleright}, p_{\triangleright}}$$</td>
</tr>
<tr>
<td>Existential formula</td>
<td>$${p_{\triangleright}, p_{\triangleright}, p_{\triangleright}, p_{\triangleright}}$$</td>
<td>$${p_{\triangleright}, p_{\triangleright}, p_{\triangleright}, p_{\triangleright}}$$</td>
<td>$${p_{\triangleright}, p_{\triangleright}, p_{\triangleright}, p_{\triangleright}}$$</td>
</tr>
</tbody>
</table>

Algorithm 1: Impact propagation algorithm

Input: $$s$$ (consistency constraint), chg (context change)

Output: impact (ultimate impact caused by chg on s)

1: impact := $$(\emptyset, \{p_{\triangleright}, p_{\triangleright}, p_{\triangleright}, p_{\triangleright}\})$$;
2: let $$F_{all}$$ be chg’s all-initial-impact formula in s;
3: for each f_{loc} in F_{all} do
4: $$(P_{pos}, P_{imp}) :=$$ chg’s initial impact on f_{loc}; // From initial impact matrix
5: let n be the syntax tree node corresponding to f_{loc};
6: while n.parent != null do
7: let f be n.parent’s corresponding formula type;
8: $$p_{pos} := \emptyset (p_{\triangleright}) \cup \ldots \cup \emptyset (p_{\triangleright}) \ p_{\triangleright} \in P_{pos};$$
9: $$p_{imp} := P_{pos};$$
10: n := n.parent
11: end while
12: impact$$_{\triangleright}$$ := impact$$_{\triangleright}$$ \cup P_{pos}; // Possible changes
13: impact$$_{\triangleright}$$ := impact$$_{\triangleright}$$ \ \setminus \ \text{impact}$$_{\triangleright}$$; // Impossible changes
14: end for
15: return impact

<addition, $$V_\triangleright$$>, its initial-impact formula is “$$\forall v_1 \in V_x \ [\ldots]$$”. We understand that when one adds an element into context $$V_x$$ all truth value changes except $$p_{\triangleright}$$ are possible for this formula. This is because if a universal formula is violated (i.e., evaluated to false), adding any element into $$V_x$$ cannot make it satisfied (i.e., evaluated to true). Similarly, given a deletion change <deletion, $$V_\triangleright$$> and its initial-impact formula “$$\forall v_1 \in V_x \ [\ldots]$$”, all truth value changes except $$p_{\triangleright}$$ are possible for this formula. This is because if a universal formula is satisfied, deleting any element from $$V_x$$ cannot make it violated. Nevertheless, for an update change, all four truth value changes are possible for a universal formula. Existential formula can be explained similarly, and we omit its discussions.

With this initial impact modeling, we in the following explain its propagation to the entire constraint.

### 3.2 Impact Propagation

The process of propagating the initial impact of a context change to an entire constraint to affect its truth value, where it becomes ultimate impact, is named impact propagation. The process works on the constraint’s corresponding syntax tree. Constructing a syntax tree for a constraint is systematic according to this constraint’s syntax hierarchy. We give two examples in Fig. 4, where the left one is for the sloc constraint and right one for the shall constraint.

Given a context change, its initial impact occurs to nodes in a syntax tree containing initial-impact formulæ. The impact is then propagated upward to the root node of the tree. For example, consider the sloc constraint and a context change <deletion, $$V_\triangleright$$> affecting it. The change’s corresponding initial-impact formula is “$$\exists v_2 \in V_y \ [\ldots]$$”, and thus its initial impact occurs to the “$$\exists v_2 \in V_y$$” node in Fig. 4 (left). The impact propagates from this “$$\exists v_2 \in V_y$$” node, its upper “not” node, and finally to the root node “$$\forall v_1 \in V_x$$”, as the arrow shows.

During this propagation, the impact visits multiple
tally six formula types that can be visited, we

\[ \forall \gamma \in C \{ f \} \]

\[
\begin{array}{ccc}
\text{(true)} & \text{(false)} & \text{(false)} \\
\forall \gamma \in C & \forall \gamma \in C & \forall \gamma \in C \\
\text{f} & \text{f} & \text{f} \\
\text{(true ... true)} & \text{(false ... false)} & \text{(true ... false)} \\
(1) & (2) & (3)
\end{array}
\]

Fig. 6. Evaluating a universal formula’s truth value (three cases).

Auxiliary function: flip, where

\[ \text{flip}(p_{\text{TT}}) := p_{\text{TF}}, \quad \text{flip}(p_{\text{TF}}) := p_{\text{TT}}, \quad \text{flip}(p_{\text{FF}}) := p_{\text{TT}}, \quad \text{flip}(p_{\text{TF}}) := p_{\text{TF}}. \]

1. \( \phi_i(p) := \{ p \} \cup \{ p_{\text{TT}} \} \).
2. \( \phi_i(p_{\text{TT}}) := \{ p \} \cup \{ p_{\text{TT}} \} \).
3. \( \phi_{\text{and}}(p) := \{ p \} \cup \{ p_{\text{TT}} \} \).
4. \( \phi_{\text{or}}(p_{\text{FF}}) := \{ p \} \cup \{ p_{\text{TT}} \} \).
5. \( \phi_{\text{implies}}(p) := (1) \{ \text{flip}(p) \} \cup \{ p_{\text{TT}} \} \). // First sub-formula case

\( (2) p \cup \{ p_{\text{TT}} \} \). // Second sub-formula case
6. \( \phi_{\text{not}}(p) := (\text{flip}(p)) \).

Fig. 7. Proposition function semantics.

nodes and can change its value due to formula semantics associated with these nodes. This works like a series of mapping from an input impact to an output impact. Since there are totally six formula types that can be visited, we model such mapping using six functions: \( \phi_i, \phi_{\text{and}}, \phi_{\text{or}}, \phi_{\text{implies}} \) and \( \phi_{\text{not}} \). They are named propagation functions. Note that there is no propagation function for terminals as they are all associated with leaf nodes. Let \( \phi \) be the propagation function of formula \( f \). Fig. 5 gives our impact propagation algorithm. We explain it in the following.

Given a context change, the algorithm initializes its impact to be \( \{ \phi \{ p_{\text{TT}}, p_{\text{TF}}, p_{\text{FF}} \} \} \), meaning that no truth value change is possible for any constraint affected by this change (Line 1). It then iterates on each initial-impact formula due to this context change, and propagates the formula’s received initial impact to the entire constraint (Lines 3–14). During the propagation, some nodes are visited, and new possible truth value changes are calculated based on propagation functions associated with these nodes (Lines 6–11). All possible truth value changes eventually merge at the root node, and impossible truth value changes are derived accordingly (Lines 12–13).

The algorithm calculates new possible truth value changes using propagation functions, which vary with different formula types. We take universal formula as an example to explain how to derive its propagation function. Consider a universal formula \( g \) given by “\( \forall \gamma \in C \{ f \} \)”, i.e., \( f \) is \( g \)’s sub-formula. Then \( g \)’s truth value is evaluated to be \( f \)’s satisﬁability by all \( \gamma \) values in context \( C \). There are three cases as illustrated in Fig. 6: (1) \( f \) is true for all \( \gamma \) values in \( C \); (2) \( f \) is false for all \( \gamma \) values; (3) \( f \) is true only for some (but not all) \( \gamma \) values. According to universal formula’s semantics, \( g \) is evaluated to true, false and false, respectively.

To derive possible truth value changes enforced by a universal formula, we consider four truth value changes in turn. Suppose that truth value change \( p_{\text{TT}} \) occurs to \( f \)’s satisﬁability by one of \( \gamma \) values. This applies only to Cases (1) and (3). Since \( p_{\text{TT}} \) does not change \( f \)’s satisﬁability, \( g \)’s truth value would remain unchanged. As a result, truth value change \( p_{\text{TT}} \) is mapped to \( \{ p_{\text{TT}}, p_{\text{TF}} \} \) as the enforcement by this universal formula (corresponding to the two cases, respectively). Similarly, truth value change \( p_{\text{FF}} \) is mapped to \( \{ p_{\text{TF}}, p_{\text{FF}} \} \). This is because \( p_{\text{TF}} \) applies only to Cases (2) and (3), and both cases have \( g \)’s truth value remain to be false. We then consider truth value change \( p_{\text{TF}} \), which applies only to Cases (1) and (3). For both cases, since \( f \) fails to hold for some \( \gamma \) values, \( g \) would be evaluated to false, i.e., leading to \( p_{\text{TF}} \) and \( p_{\text{FF}} \), respectively, for the two cases. As a result, truth value change \( p_{\text{TF}} \) is mapped to \( \{ p_{\text{TF}}, p_{\text{FF}} \} \). Finally, truth value change \( p_{\text{TF}} \) applies only to Cases (2) and (3). This indicates that \( f \)’s truth value changes from false to true for one \( \gamma \) value. Then \( g \)’s truth value can either change to true or remain to be false, depending on whether or not \( f \) now becomes true for all \( \gamma \) values in \( C \). As a result, truth value change \( p_{\text{TF}} \) is mapped to \( \{ p_{\text{TF}}, p_{\text{FF}} \} \). Combining all these possibilities, a universal formula’s propagation function is given as follows:

\[ \varphi(p) := \begin{cases} (1) \{ p_{\text{TF}}, p_{\text{FF}} \}, & \text{if } p = p_{\text{TT}}; \\
(2) \{ p_{\text{TF}}, p_{\text{FF}} \}, & \text{if } p = p_{\text{TF}}; \\
(3) \{ p_{\text{TF}}, p_{\text{FF}} \}, & \text{if } p = p_{\text{FF}}; \\
(4) \{ p_{\text{FF}} \}, & \text{if } p = p_{\text{FF}}. 
\end{cases} \]

It can be shortened as:

\[ \varphi(p) := \{ p \} \cup \{ p_{\text{TF}} \}. \]

The other five propagation functions can be derived similarly, and we give their details in the Appendix. Fig. 7 gives all propagation function semantics. For example, a universal formula’s propagation function is: “\( \varphi(p) := \{ p \} \cup \{ p_{\text{TF}} \} \)”, which is exactly what we derived. It means that if a truth value change passes by a universal formula in its propagation, \( p_{\text{FF}} \) becomes also possible unless the original truth value change is already \( p_{\text{FF}} \). “\( \phi_{\text{implies}}(p) \)” has two cases, depending on which sub-formula an input truth value change is propagated to. Since an “implies” formula has two sub-formulae, to distinguish them, we name the one before the “implies” operator first sub-formula, and the other one second sub-formula. An auxiliary function “flip” is introduced for inverting truth value changes, e.g., from \( p_{\text{TT}} \) to \( p_{\text{FF}} \), from \( p_{\text{TF}} \) to \( p_{\text{TF}} \), and so on.

We give two examples in Fig. 8. The first example

\[ \forall v \in V, \exists v \in V, \exists g \in G
\]

\[ \{ (p_{\text{TT}}, p_{\text{TF}}, p_{\text{FF}}), \} \quad \{ (p_{\text{TF}}, p_{\text{TT}}, p_{\text{FF}}), \} \quad \{ (p_{\text{TF}}, p_{\text{TF}}, p_{\text{FF}}), \} 
\]

(ultimate impact)

(initial impact)

\[ \text{true} \quad \text{g.loc} = \text{"hall s"} \]

Fig. 8. Two propagation examples (top: for \( s_{\text{not}} \); bottom: for \( s_{\text{mat}} \).
Algorithm 2: Constraint instability analysis algorithm

**Input:** \( s \) (consistency constraint)

**Output:** \( ICS \) (instability condition set for \( s \))

1. \( ICS := \emptyset \)
2. **for** each context \( C_i \) referenced by \( s \) **do**
3. **for** each context \( C_j \) referenced by \( s \) **do** // \( C_j \) can be \( C_i \)
4. **for** each combination \( \langle \text{type}_1, C_i \rangle \oplus \langle \text{type}_2, C_j \rangle \), where \( \text{type}_1, \text{type}_2 \in \{+,-,\#\} \) **do** // Nine combinations
5. **if** \( \langle \text{type}_1, C_i \rangle \)’s ultimate impact includes \( p_{\text{TF}} \) and \( \langle \text{type}_2, C_j \rangle \)’s ultimate impact includes \( p_{\text{TF}} \) **then**
6. \( ICS := ICS + \{ \langle \text{type}_1, C_i \rangle \oplus \langle \text{type}_2, C_j \rangle \} \)
7. **end if**
8. **end for**
9. **end for**
10. **end for**
11. return \( ICS \)

Fig. 9. Constraint instability analysis algorithm.

shows how a deletion change \( \langle \text{deletion}, V_i \rangle \) impacts the \( s_{\text{loc}} \) constraint. The change generates its initial impact at the “\( \exists d_i \in V_y \)” node (containing the “\( \exists d_i \in V_y (...) \)” initial-impact formula). The impact propagates through its upper “not” node and arrives at the root node “\( \forall d_i \in V_x \)”.

This deletion change’s ultimate impact is: \( \langle [p_{\text{TF}}, p_{\text{TF}}, p_{\text{TF}}, p_{\text{TF}}]; \emptyset \rangle \). It means that the change can never alter this constraint’s truth value from true to false, while other truth value changes are possible. The second example shows how an update change \( \langle \text{update}, G \rangle \) impacts the \( s_{\text{hall}} \) constraint. The change generates its initial impact at the “\( \exists g \in G \)” node (containing the “\( \exists g \in G [...] \)” initial-impact formula). After propagation, the change’s ultimate impact is: \( \langle [p_{\text{TF}}, p_{\text{TF}}, p_{\text{TF}}, p_{\text{TF}}]; \emptyset \rangle \). It means that all truth value changes are possible for this constraint. We use such impact information to analyze a constraint’s instability to the detection of STINs as we explain the following.

3.3 Instability Analysis

In our instability analysis, we are interested in two truth value changes: \( p_{\text{TF}} \) and \( p_{\text{TF}} \). The former refers to a constraint’s truth value changing from true to false, indicating the detection of a new context inconsistency. The latter refers to a constraint’s truth value changing from false to true, indicating the vanishing of a previously detected context inconsistency.

Let \( \text{chg}_1 \) and \( \text{chg}_2 \) be two context changes collected in turn, and \( s \) be a consistency constraint affected by them. If \( \text{chg}_1 \)’s ultimate impact on \( s \) contains \( p_{\text{TF}} \) and \( \text{chg}_2 \)’s contains \( p_{\text{TF}} \) as possible truth value change, then scheduling inconsistency detection between \( \text{chg}_1 \) and \( \text{chg}_2 \) can be subject to STIN. This is exactly our earlier discussed situation: \( \text{check}(P_1, s) = \text{false} \) and \( \text{check}(P_2, s) = \text{true} \), where \( P_1 := \text{apply}(P_0, \text{chg}_1) \), \( P_2 := \text{apply}(P_0, \text{chg}_2) \), and \( P_0 \) is the context pool before applying the two context changes. We name such two context changes an instability condition. A consistency constraint can have multiple instability conditions. If a pair of context changes matches any of them, checking this constraint between applying the two changes can be subject to the detection of STINs. Note that the two context changes may not necessarily be consecutive, and other context changes can be in between. In that case, detected STINs would have a window of size larger than one, i.e., vitality-\( k \) STINs (\( k \geq 2 \)), as discussed earlier in Subsection 2.2. We will come back to this issue again later in Section 4.

One can derive all instability conditions from a consistency constraint by examining the constraint in a systematic way. Suppose that a constraint references\( n \) distinct contexts. Each context can be changed in three ways: addition, deletion or update change. Then there are a total of \( 9n^2 \) (\( = 3n \times 3n \)) combinations for a pair of context changes. Fig. 9 gives our constraint instability analysis algorithm. It examines all these combinations to identify instability conditions. For ease of presentation, we use symbols “+”, “−” and “#” to represent “addition”, “deletion” and “update”, respectively, and use “⊕” to connect two context changes.

The algorithm examines the \( 9n^2 \) combinations, identifies instability conditions, and stores them into the \( ICS \) set as the return result. The algorithm has a time complexity of \( O(9n^2) = O(n^2) \), where \( n \) is the number of distinct contexts referenced by the constraint under analysis. In practice, \( n \) is typically small (e.g., 2 for constraints \( s_{\text{loc}} \) and \( s_{\text{hall}} \)). Thus the time cost for analyzing a constraint’s instability is usually very small (say, several milliseconds), as our later evaluation shows.

We give two instability analysis examples. Consider the \( s_{\text{hall}} \) constraint: \( \forall d_i \in V_x \) [true] implies \( \exists g \in G \) \( g.loc = \text{"hall s"} \]). The constraint references two contexts: \( V_x \) and \( G \) and thus it has a total of \( 9 \times 2^2 = 36 \) combinations for examination. The analysis algorithm returns 16 combinations out of them as instability conditions (“+/#” means “+” or “#”):

\[
\begin{align*}
&<+/#, V_x \rangle \ominus <-/#, V_x >, \\
&<+/#, V_x > \ominus <+/#, G >, \\
&<-/#, G > \ominus <-/#, V_x >, \\
&<-/#, G > \ominus <+/#, G >.
\end{align*}
\]

Consider another \( s_{\text{loc}} \) constraint: \( \forall d_i \in V_x \) [not (\( \exists d_i \in V_y \) \( d_i.id = d_j.id \))]. It also references two contexts, and thus has 36 combinations for examination. The algorithm also returns 16 but different instability conditions:

\[
\begin{align*}
&<+/#, V_x > \ominus <-/#, V_x >, \\
&<+/#, V_x > \ominus <+/#, V_y >, \\
&<+/#, V_y > \ominus <+/#, V_x >.
\end{align*}
\]

These results are based on theoretical analysis only. In practice, some context changes may not be available. For example, in our exhibition application there are only addition and deletion changes for contexts \( V_x, V_y, V_y \), and only update changes for context \( G \). As such, some instability conditions should be pruned. After pruning, four instability conditions result for the \( s_{\text{hall}} \) constraint:

\[
\begin{align*}
&<+, V_x > \ominus <-, V_x >, \\
&<+, V_x > \ominus <#, G >, \\
&<#, G > \ominus <-, V_x >, \\
&<#, G > \ominus <#, G >.
\end{align*}
\]

Similarly, four instability conditions result for the \( s_{\text{loc}} \) constraint:

\[
\begin{align*}
&<+, V_x > \ominus <-, V_x >, \\
&<+, V_x > \ominus <-, V_y >, \\
&<+, V_y > \ominus <-, V_x >, \\
&<+, V_y > \ominus <-, V_y >.
\end{align*}
\]

Such instability conditions are useful in guiding smart scheduling of inconsistency detection to suppress the detection of STINs. Let us revisit our exhibition application.
Fig. 10. Inconsistency detection function.

Algorithm 3: Immediate scheduling algorithm (IMD)
Global variables: P (context pool), S (all consistency constraints)
Input: chg, (new context change)
1: P := apply(P, chg);
2: for each constraint s ∈ S do
3: if chg affects s then // s references a context concerned by chg,
4: detect(P, s) // Inconsistency detection
5: end if
6: end for
7: return

Fig. 11. Immediate scheduling algorithm.

In Scenario 1 (Fig. 1), changes “v1 enters” and “g enters” match instability condition \(<\!, \ V\!\!>\oplus\!\!\#\!, \ G\!\!>,\) and thus inconsistency detection should not be performed between them; changes “g enters” and “v2 enters” do not match any instability condition, and thus inconsistency detection should be performed between them; similarly, inconsistency detection should be performed between changes “v2 enters” and “v3 enters”. This is exactly the desirable schedule suggested in Fig. 1. In Scenario 2 (Fig. 2), changes “entering y” and “leaving x” match instability condition \(<\!, \ V\!\!>\oplus\!\!\#\!, \ V\!\!>,\) and thus inconsistency detection should not be performed between them; change “leaving x” does not match the first part of any instability condition, and thus inconsistency detection should be performed upon this change. This is also exactly the desirable schedule suggested in Fig. 2. Note that these examples are only simple cases (i.e., suppressing the detection of vitality-1 STINs). We study more complicated cases in the following.

4 Suppressing the Detection of STINs

In this section, we present different algorithms for scheduling inconsistency detection to suppress the detection of STINs. We start with an immediate scheduling algorithm, which is commonly used in existing inconsistency detection practice. We then propose our selective scheduling algorithms, which have several variants with different trade-offs. They are all based on our derived instability conditions.

4.1 Immediate Scheduling

The immediate scheduling algorithm (IMD) performs inconsistency detection upon each context change (i.e., whenever a new context change is collected). It is the traditional way of detecting inconsistencies eagerly. IMD is subject to the detection of STINs as we discussed earlier. It works as a baseline for our comparison.

We define in Fig. 10 an inconsistency detection function (detect) for the IMD algorithm. The function checks whether the contexts stored in the context pool are consistent with respect to a given consistency constraint, and if not, would calculate and report resulting context inconsistencies. The detailed inconsistency calculation process is not our focus in this article, and therefore omitted. Interested readers can refer to existing work [38], [39] and our previous work [61] for a detailed explanation and examples. With this function, we present the IMD algorithm in Fig. 11. IMD works intuitively by checking every constraint for inconsistency if a collected new context change affects it.

### 4.2 Selective Scheduling

We then discuss how to use our derived instability conditions to suppress the detection of STINs. This is achieved by scheduling inconsistency detection selectively. Here, “selectively” means that upon each collected context change, one would decide whether or not to perform inconsistency detection for this change (thus named selective scheduling).

We first introduce a basic version of this decision process. Consider a consistency constraint \(s\) and a collected context change \(chg\). If change \(chg\) does not affect constraint \(s\), there is no need to perform inconsistency detection with \(s\). Otherwise, change \(chg\) affects constraint \(s\) and one has to examine its impact. There are two cases. (1) If there is no instability condition from constraint \(s\) such that its first part matches change \(chg\), one can safely perform inconsistency detection with \(s\) for \(chg\). This is because change \(chg\) will never match any instability condition with later context changes. (2) Otherwise, change \(chg\) is suspicious and needs further examination. When the next change \(chg_2\) is collected, one can know whether the pair, \(chg_1\opluschg_2\), matches any instability condition from constraint \(s\). There are also two cases. (1) If the answer is yes, the inconsistency detection with constraint \(s\) should not be performed for change \(chg_1\), because otherwise STINs may be detected. (2) If the answer is no, the inconsistency detection with constraint \(s\) should be performed for change \(chg_1\), because otherwise stable context inconsistencies may be missed. This is the basic decision process, for one constraint and for one pair of context changes of a window of size one (i.e., two context changes are consecutive).

To extend this decision process to support multiple constraints, one needs a sliding-window data structure as illustrated in Table 2. In the table, \(s\) is a constraint under consideration on whether to perform inconsistency detection with it. Adding more rows allows considering more constraints at the same time. The first row specifies two consecutive context changes for examination. They form a window of size one (extension of supporting a larger size discussed later in Subsection 4.3). The cells under these changes indicate whether or not to perform inconsistency detection with respect to a certain constraint. The cell values can be “\(\checkmark\)”, “\(\times\)” and “\(?\)”, where “\(\checkmark\)” means “should

<table>
<thead>
<tr>
<th></th>
<th>chg1</th>
<th>chg2</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>(\times)</td>
<td>(\checkmark)</td>
<td>...</td>
</tr>
</tbody>
</table>

### Table 2. Sliding-window data structure.
Fig. 12. Decision process illustration.

perform”, “x” means “should not perform”, and “?” means “not decided yet”. For example, Table 2 indicates that upon context change chgi, one should not perform inconsistency detection with constraint s, but should for change chgi+1.

We illustrate this decision process in Fig. 12. It shows that the decision of whether to perform inconsistency detection with constraint s for change chgi can be made once chgi is collected, or at a later time when the next change chgi+1 is collected. Thus the scheduling decision is made individually for every constraint. For example, when change chgi is collected, one may decide immediately to perform or not to perform inconsistency detection with constraint s1 for chgi, but may need more information to decide for another constraint s2. Nevertheless, for this version of selective scheduling, the decision can be delayed at most for a window of size one (i.e., the next context change), and in most cases has no delay (illustrated in Fig. 12 and also confirmed in our later evaluation). When scheduling decisions are made for all constraints with respect to change chgi, the sliding window proceeds by one step.

We present our selective scheduling algorithm (SEL) in Fig. 14. It performs inconsistency detection selectively based on instability conditions derived from consistency constraints to suppress the detection of STINs. The algorithm consists of two parts (Lines 1–13 and 14–25). The first part finalizes scheduling decision for each constraint s with respect to its last context change s.pending if necessary. When the decision is made, the inconsistency detection with constraint s would be performed or not, depending on the actual decision value (Lines 3–8). Note that the decision is made individually for each constraint. Therefore, each constraint s needs to maintain a reference to its last context change s.pending, so that inconsistency detection can be performed on top of “apply(P, s.pending)” if necessary. When all constraints have handled this context change, i.e., each s.pending becomes s.done, the change can be permanently applied to P. This is what function “flush” (defined in Fig. 13) does (Line 12). The second part of the algorithm makes scheduling decision for each constraint with respect to the current context change chgi if possible (Lines 15–22), according to our explained decision process (Fig. 12).

SEL supports making scheduling decisions for multiple constraints with respect to context change pairs of a window of size one. It suppresses the detection of vitality-1 STINs. We in the following extend it to support suppressing the detection of vitality-n STINs (n ≥ 1).

4.3 Vitality-based Selective Scheduling

The further extended algorithm is named vitality-based selective scheduling (VIT). It performs inconsistency detection selectively to suppress the detection of STINs with a vitality of no more than n (n ≥ 1). The preceding SEL algorithm is actually a special case of the VIT algorithm (i.e., when n = 1).

We present the VIT algorithm in Fig. 16. It is also named VIT, for its conservative strategy (we discuss this strategy and present another aggressive strategy later). VIT resembles SEL. It also consists of two parts, for finalizing scheduling decisions for past context changes (Lines 1–14) and for deciding scheduling for the current context change (Lines 15–26), respectively. The only difference lies in the monitored sliding window, which increases from size one to n. When the current context change chgi is collected, SEL examines only one past context change chgi-1, which is stored in s.pending for each constraint s. For VIT, n past context changes, chgi-1 to chgi-n, may be examined. They are stored in a queue s.pending for every constraint s. Similarly, its s.done is also based on a queue data
Fig. 15. The flush<sub>h</sub> function.

Algorithm 6: Vitality-based selective scheduling algorithm (VIT<sub>v</sub>)

4: \( \text{cell}(s, \text{chg}) := \text{"x"}; \text{done}(\text{s.pending.out}) \)  

Changed to

4.1: \( \text{while \s .pending.empty() do} \)  
4.2: \( \text{cell}(s, s . pending . head()) := \text{"x"}; \text{done}(\text{s.pending.out}) \)  
4.3: \( \text{end while} \)

Fig. 17. Vitality-based selective scheduling algorithm (VIT<sub>v</sub>.

structure, for containing handled context changes according to their temporal order. For ease of presentation, we extend the preceding “apply” function to allow it to accept a queue of context changes. Then all changes in the queue will be applied to context pool \( P \) in turn (Lines 7 and 22). The preceding flush<sub>h</sub> function is also extended to flush<sub>h</sub> (defined in Fig. 15), which applies permanently those context changes (at most \( n \)) that have been handled by all constraints to context pool \( P \) in turn.

We note that the VIT<sub>v</sub> algorithm actually takes a conservative strategy. When \( n \) is larger than one, more than one past context change can be examined in the sliding window. The scheduling decisions for these past context changes may conflict with each other. For example, suppose that pair \( chg_{i,k} \oplus chg_{i} \) (1 \( \leq k \leq n \)) matches an instability condition but pair \( chg_{i,k+1} \oplus x \) does not (i.e., no \( x \) can make \( chg_{i,k+1} \oplus x \) belong to \( s.ICS \)), with respect to a constraint \( s \). According to the VIT<sub>v</sub> algorithm in Fig. 16, \( \text{cell}(s, \text{chg}_{i,k}) \) is set to “\( n \)” (Line 4) but \( \text{cell}(s, \text{chg}_{i,k+1}) \) is set to “\( n \)” (Line 21). We observe that the two scheduling decisions are conflicting with each other. In this example, \( \text{cell}(s, \text{chg}_{i,k}) \) suggests that change \( chg_{i,k} \) matches an instability condition with change \( chg_{i} \) and thus inconsistency detection should not be performed between them (otherwise STINs may be detected). However, \( \text{cell}(s, \text{chg}_{i,k+1}) \) suggests that inconsistency detection should be performed when change \( chg_{i,k+1} \) is collected (otherwise stable context inconsistencies may be missed). Since \( chg_{i,k+1} \) is between \( chg_{i,k} \) and \( chg_{i} \), the conflict arises. To resolve it, the VIT<sub>v</sub> algorithm takes a conservative strategy to give priority to \( \text{cell}(s, \text{chg}_{i,k+1}) \). This can avoid false negatives (i.e., preserving the detection of stable inconsistencies), but few false positives may result (i.e., few STINs may remain). Our vitality-based selective scheduling can also take an aggressive strategy to give priority to \( \text{cell}(s, \text{chg}_{i,k}) \). This can avoid false positives (i.e., avoiding the detection of STINs), but few false negatives may result (i.e., few stable inconsistencies may be missed). The two strategies represent two ends of application requirements: (1) conservative strategy gives zero tolerance to any missing of stable context inconsistencies, and (2) aggressive strategy gives zero tolerance to any detection of STINs.

By taking the above aggressive strategy, we present another vitality-based selective scheduling algorithm VIT<sub>a</sub>.

Algorithm 5: Vitality-based selective scheduling algorithm (VIT<sub>a</sub>)

Global variables: \( P \) (context pool), \( S \) (all consistency constraints), \( cells \) (sliding window of size \( n \))

Input: \( chg \), (new context change)

1: \( \text{for each constraint} s \in S \text{ do} \) // Finalize scheduling decisions
2: \( \text{while} s . pending . empty() \text{ do} \)
3: \( \text{if} s . pending . head() \oplus chg \in s . ICS \text{then} \)
4: \( \text{if} s . pending . head() := "x"; s . done.in(s . pending . out()) \)
5: \( \text{else if} s . pending . head() \Rightarrow \text{then} \) // Reaching boundary
6: \( \text{if} s . pending . head() := "x"; s . done.in(s . pending . out()) \)
7: \( \text{detect}(\text{apply}(P, s . done), s) \) // Check \( s \) against \( P \) and \( s . done \) together
8: \( \text{else} \)
9: \( \text{break} \)
10: \( \text{end if} \)
11: \( \text{end while} \)
12: \( \text{end for} \)

13: \( \text{flush}(y) \); // Flush past context changes into context pool if possible
14: \( \text{cells.lower :=} i - n + 1; \Rightarrow +1 \text{ to sliding-window’s lower boundary} \)
15: \( \text{for each constraint} s \in S \text{ do} \) // Decide scheduling for \( chg \) if possible
16: \( \text{if} chg \text{ does not affect} s \text{ then} \) // \( s \) does not reference any context in \( chg \),
17: \( \text{cell}(s, chg) := "x"; s . done.in(chg) \)
18: \( \text{else if} \exists x \text{ such that} chg \oplus x \in s . ICS \text{then} \)
19: \( \text{cell}(s, chg) := "x"; s . pending.in(chg) \) // Decide later
20: \( \text{else} \)
21: \( \text{cell}(s, chg) := "x"; s . done.in(chg) \);
22: \( \text{detect}(\text{apply}(P, s . done), s) \) // Check \( s \) against \( P \) and \( s . done \) together
23: \( \text{end if} \)
24: \( \text{end for} \)
25: \( \text{flush}(y) \); // Flush \( chg \) into context pool if possible
26: \( \text{cells.upper :=} i + 1; \Rightarrow +1 \text{ to sliding-window’s upper boundary} \)
27: \( \text{return} \)

Fig. 16. Vitality-based selective scheduling algorithm (VIT<sub>a</sub>.

in Fig. 17. It only modifies Line 4 in VIT<sub>a</sub> to Lines 4.1–4.3 in VIT<sub>v</sub>. For easier reading, we list only modified lines in Fig. 17. Both the VIT<sub>v</sub> and VIT<sub>a</sub> algorithms belong to vitality-based selective scheduling. They have different trade-offs in suppressing the detection of STINs. Besides, one may also derive algorithm variants compromised from them for balancing the trade-off. We measure and compare these algorithms and one variant in later evaluation about their effectiveness in suppressing the detection of STINs as well as in preserving the detection of stable context inconsistencies.

5 Evaluation

In this section, we introduce the implementation of our constraint instability analysis and STIN suppression, and evaluate the effectiveness and trade-offs of different STIN suppression algorithms.

5.1 Implementation

Our implementation contains two parts. One is a constraint instability analysis tool, named CINA, and the other is a plug-in for integration with context middleware that handles context changes for applications. We built them in Java 7.

CINA tool. The CINA tool is used for editing consistency constraints and deciding whether these constraints are subject to the detection of STINs. If yes, it would derive a set of instability conditions for each of such constraints. These instability conditions can be used by the middleware plug-in to suppress the detection of STINs during its context inconsistency detection.
CINA consists of three components, namely, EDITOR, VERIFIER and ANALYZER, as illustrated in Fig. 18. The EDITOR allows its users to edit contexts (Area A), context change types (Area B) and consistency constraints (Areas C, D and E). The VERIFIER checks edited constraints for syntactic and semantic errors. The ANALYZER decides whether these constraints are subject to the detection of STINs (Area F) and derives their instability conditions if any (Areas G and H).

The CINA tool is handy and efficient. Its Java source files contain 22 packages, 85 classes, and approximately 4,300 lines of code without white space and comments. In our following evaluation, CINA derived 113 instability conditions from 29 consistency constraints in 36 ms (or 1.24 ms per constraint). This is highly efficient.

**Middleware plug-in.** The plug-in is for integration with a middleware infrastructure that handles context changes for applications. When the middleware receives context changes from its connected sensors, which can be heterogenous and distributed, it passes the changes to the plug-in before it applies them to its context pool, and later feeds contexts in the pool to its upper-layer applications. The plug-in detects context inconsistencies according to prespecified consistency constraints, and resolves them according to application requirements or predefined policies. It exploits the CINA tool’s derived instability conditions to suppress the detection of STINs during its inconsistency detection.

For our experimental purposes, we integrated the plug-in into our previous context middleware, Cabot [58], [61] for connecting to sensors and applications. The plug-in allows using different scheduling algorithms in its context inconsistency detection. Its Java source files contain 8 packages, 50 classes, and approximately 2,300 lines of code without white space and comments.

### 5.2 Experimental Setup

We experimentally compare different inconsistency detection scheduling algorithms in suppressing the detection of STINs and in preserving the detection of stable context inconsistencies. The IMD algorithm is used as the baseline for measuring stable inconsistencies and STINs in experimental subjects. Our Sel and Vit algorithms can both suppress the detection of STINs. Since Sel is a special case of Vit (Vit becomes Sel when its vitality parameter is set to 1), we evaluated Vit only in our experiments. Vit has two algorithm variants Vit_l and Vit_r. They represent two extreme ends: Vit_l disables inconsistency detection scheduling for change \( \text{chg}_{i+n} \) only once \( \text{chg}_{i+n} \) and \( \text{chg}_i \) match any instability condition, while Vit_r disables scheduling from changes \( \text{chg}_{i+n} \) to \( \text{chg}_i \) (0 \( \leq k \leq n - 1 \)). In principle, one can derive many new algorithm variants from them by disabling inconsistency detection scheduling from change \( \text{chg}_{i+n} \) but not to change \( \text{chg}_i \) yet. Such variants can be useful in balancing the ability of suppressing the detection of STINs and the limitation of missing the detection of stable context inconsistencies. For comparison purposes, we selected one algorithm variant Vit_l as the representative of this balancing, which disables inconsistency detection scheduling from changes \( \text{chg}_{i+n} \) to \( \text{chg}_i \), i.e., in half.

We selected three real-world applications as our experimental subjects. The first subject is our aforementioned Exhibition application, which was deployed in our departmental building. The application concerns three exhibition halls and one connecting corridor. Each hall door is installed with an RFID reader to detect visitor entering or leaving events. The second application, Smart Light [50], works for a warehouse scenario, in which several lights cooperatively provide illumination for warehouse workers. It aims to guarantee each worker receiving necessary illumination and reduce the overall power consumption. It keeps monitoring workers in each light’s illumination range. If there is no worker near a light, it would reduce the light’s power gradually until the minimal level. Otherwise, it would measure each nearby worker’s received illumination. If the illumination is not enough, it would increase the light’s power gradually until the illumination has become enough or the power has reached the maximal level. Each light can cover several workers and each worker can be covered by several lights, and therefore the whole illumination cooperation can be complex. This Smart Light application is from a third party [50], and we deployed it in a paper company’s warehouse in our RFID-enabled task automation study [62]. The application detects workers by infrared sensing and the illumination is measured by light sensors.

We conducted controlled experiments with these two applications in a laboratory setting but with real deployment. The real deployment enables us to collect real data as well as observing practical STIN issues. The laboratory setting enables us to control key parameters (e.g., vitality)
and repeat experiments to compare the effectiveness of different scheduling algorithms. While the experiments with the two applications might seem small-scale, we also conducted another large-scale case study with the third application, Smart City [61], which handled over 1.5 million real taxi data. We discuss this case study later in Subsection 5.7.

We in the following present experimental results with the first two applications. The experiments were conducted on a machine with an Intel® Core™ i5 CPU @3.2GHz and 4GB RAM. The machine is installed with MS Windows 7 Professional and Oracle Java 7. The two applications used 6 and 7 consistency constraints, respectively, for inconsistency detection and handling. For the Exhibition application, these constraints are for ensuring location consistency, guide accompanying requirement, and buddy system requirement (some exhibition contents need assistance of two visitors as a buddy system for demo purposes). We name these three categories of constraints Type-loc, Type-gud and Type-bud constraints, respectively. For the Smart Light application, these constraints are for ensuring the overall coverage (some lights can cover the whole warehouse), equal coverage (some lights share the same illumination ranges), subsumed coverage (some lights have their illumination ranges subsumed in those of other lights), and disjoint coverage (some lights have their illumination ranges disjointed with each other). We name these four categories Type-ovr, Type-eqv, Type-sub and Type-dsj constraints, respectively. We used our CINA tool to derive instability conditions from these 13 consistency constraints, and obtained a total of 52 instability conditions (24 and 28 for the two applications, respectively). Table 3 lists the statistics information about these constraints, as well as their used formula types (all seven formula types have been covered).

With these experimental subjects and consistency constraints, we study the following research questions:

**RQ1:** Are STINs common in context inconsistencies detected for context-aware applications?

**RQ2:** Are our scheduling algorithms effective in suppressing the detection of STINs and in preserving the detection of stable context inconsistencies?

**RQ3:** What is the runtime overhead of our scheduling algorithms?

**RQ4:** How do our scheduling algorithms compare to the aforementioned ways of avoiding reporting STINs?

We in the following four subsections (5.3, 5.4, 5.5 and 5.6) answer these four research questions in turn.

### 5.3 RQ1: STIN Fact

We ran the Exhibition application continuously for four days, during which several groups of university visitors and undergraduate students visited the concerned exhibition halls. We also ran the Smart Light application continuously for three days in our aforementioned warehouse, in which ten workers conducted their tasks and walked from time to time in the warehouse. For these two applications, we collected nearly 200 thousand and 39 thousand context changes (visitor entering and leaving events and walker movement events), respectively. From these context changes, we detected a total of 70,166 and 1,176 context inconsistencies for the two applications, respectively.

We observe that the STIN percentage for the two applications is 32.9% (23,102) and 64.3% (750), respectively, as illustrated in Fig. 19 (top) and Fig. 20 (top). Thus STINs seem quite common for the two applications: about one third to two thirds of detected context inconsistencies are STINs. These percentages are calculated with respect to all consistency constraints. We also calculate STIN percentage with respect to each constraint type, as illustrated in Fig. 19 (bottom) and Fig. 20 (bottom). We observe that the STIN percentage varies greatly with different constraint types. For example, the STIN percentage is 25.8%, 31.6% and 72.3% for Type-loc, Type-gud and Type-bud constraints, respectively, as illustrated in Fig. 19 (bottom) for the Exhibition application. It is 59.6%, 71.2%, 30.7% and 80.0% for Type-ovr, Type-eqv, Type-sub and Type-

### TABLE 3. STATISTICS INFORMATION ABOUT CONSISTENCY CONSTRAINTS USED IN EXPERIMENTS.

<table>
<thead>
<tr>
<th>Application</th>
<th>Consistency constraints (#)</th>
<th>Used formula types</th>
<th>Instability conditions (#)</th>
<th>Analysis time (ms)</th>
<th>Constraint types (for reference later)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exhibition</td>
<td>6</td>
<td>5 (\lor, \exists, implies, not, predicate)</td>
<td>24</td>
<td>6</td>
<td>Type-loc, Type-gud, Type-bud</td>
</tr>
<tr>
<td>Smart Light</td>
<td>7</td>
<td>6 (\exists, and, or, implies, not, predicate)</td>
<td>28</td>
<td>9</td>
<td>Type-ovr, Type-eqv, Type-sub, Type-dsj</td>
</tr>
<tr>
<td>Smart City</td>
<td>16</td>
<td>6 (\lor, \exists, and, implies, not, predicate)</td>
<td>61</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>29</td>
<td>All 7 formula types</td>
<td>113</td>
<td>36</td>
<td></td>
</tr>
</tbody>
</table>

![Fig. 19. STIN percentages for the Exhibition application (top: for all consistency constraints; bottom: for different constraint types).](image1)

![Fig. 20. STIN percentages for the Smart Light application (top: for all consistency constraints; bottom: for different constraint types).](image2)
dsj constraints, respectively, as illustrated in Fig. 20 (bottom) for the Smart Light application. Since these constraints have different syntax trees and formula types, it is understandable for their large difference in subject to the detection of STINs. Besides, this also makes these constraints good candidates for our STIN suppression study (in terms of constraint representativeness), by covering a wide range of STIN percentage from 25.8% to 80.0%.

A closer study discloses that these STINs have different vitalities. They happen to both fall into the range from 1 and 5. We note that this may not hold for other applications (e.g., our later Smart City application’s detected STINs have a vitality ranging from 1 to 2). Although the two applications detected STINs of the same vitality range, they have quite different vitality distributions. For the Exhibition application, its STIN percentage decreases from 10.9%, 7.9%, 5.8%, 5.2% to 3.1% for vitality-1, vitality-2, ... and vitality-5 STINs, respectively, as illustrated in Fig. 21 (top). These percentages are calculated against all context inconsistencies, and thus accumulate for a total of 32.9%, as mentioned earlier. For this application, STINs of smaller vitalities seem to occur more likely and the decreasing trend is steady. For the Smart Light application, its STIN percentage decreases rapidly from 41.0%, 12.0%, 5.7%, 3.2% to 2.4% for vitality-1, vitality-2, ... and vitality-5 STINs, respectively, as illustrated in Fig. 22 (top). Similarly, these percentages accumulate for a total of 64.3%, as explained earlier. For this application, STINs of smaller vitalities also occur more likely but the decreasing trend is rather sharp. A majority of its STINs is vitality-1 STINs (41.0% / 64.3% = 63.8%), which dominate in all STINs. Their absolute percentage is at least 3.4 times that of STINs of any other vitality value (up to 17.1 times). This difference in the vitality distribution also suggests that the consistency constraints used in the two applications have different nature, and it would be interesting to see how such diverse STINs can be suppressed by our scheduling algorithms.

We also recalculate these percentages with respect to different constraint types, as illustrated in Fig. 21 (bottom) and Fig. 22 (bottom) for the two applications, respectively. We also observe similar decreasing trends in STIN percentages for each constraint type, although their absolute values can differ largely (up to 7.7 times). Such large value differences further validate different nature of our studied consistency constraints as well as their representativeness.

As a summary, we conclude our answer to research question RQ1 as follows:

STINs are common in detected context inconsistencies. Their percentages can vary greatly with different applications, consistency constraints or vitality values.

5.4 RQ2: STIN Suppression

We next evaluate our inconsistency detection scheduling algorithms’ effectiveness in suppressing the detection of STINs and in preserving the detection of stable context inconsistencies. Our \(\text{VIT}_o\), \(\text{VIT}_b\), and \(\text{VIT}_r\), algorithms take a window size as the parameter. When the window size is set to \(n\), STINs of vitality no more than \(n\) are suppressed in inconsistency detection. Since all detected STINs have a vitality range from 1 to 5, a window value of 5 would suffice to suppress the detection of all the STINs. For a finer control, we gradually increased the window size from 1 to 5 and observed how these STINs are suppressed in inconsistency detection. Note that with the growth of the window size, our scheduling algorithms would have an increasing number of STINs to suppress. For example, for the Exhibition application when we increase the window size from 1 to 5 with a pace of 1, our scheduling algorithms need to suppress the detection of increasing STINs, from 10.9%, 18.8%, ... to 32.9%, as against all context inconsistencies (Fig. 21). For the Smart Light application, the percentage would increase from 41.0%, 53.0%, ... to 64.3%, even higher than those in the Exhibition application.

We compare the effectiveness of the three scheduling algorithms in Fig. 23 for the Exhibition application and in Fig. 24 for the Smart Light application. The leftmost bars show how many STINs need to be suppressed (they are from the IMD algorithm). Right six bars compare the three scheduling algorithms’ effectiveness in suppressing the detection of STINs and in preserving the detection of stable context inconsistencies, respectively. Some bars are invisible as they represent zero, but we still provide data labels. These three algorithms are also compared under different window sizes for disclosing their effectiveness trends.

The aggressive algorithm \(\text{VIT}_r\) aims to maximize the suppression of detected STINs. We observe that it com-
completely suppressed the detection of all STINs (i.e., STINs reduced by 100%) for both applications. As a cost of this success, it missed some stable context inconsistencies (i.e., false negatives), e.g., 0–4.9% for the Exhibition application and 0–11.5% for the Smart Light application. As such, it suits application scenarios that have zero tolerance to any STINs but allow missing of a few stable context inconsistencies, e.g., when any wrong exception handling for misreported inconsistencies is unacceptable.

The conservative algorithm VIT\textsubscript{c} aims to maximize the preservation of stable context inconsistencies during its suppression of detected STINs. We observe that it completely preserved all stable context inconsistencies (i.e., no false negative) for both applications. As a cost of this protection, it still detected some remaining STINs (i.e., false positives), e.g., 0–8.0% for the Exhibition application and 0–6.9% for the Smart Light application. Compared with the IMD algorithm, it still reduced the detection of STINs significantly, e.g., by 75.7–100% for the Exhibition application and by 89.3–100% for the Smart Light application. As such, it suits application scenarios that do not allow any missing of stable context inconsistencies but can tolerate a few STINs, e.g., when the exception handling for context inconsistencies is for safety and it does not harm anything if conducted more than necessary.

The balanced algorithm VIT\textsubscript{b} aims for a trade-off between VIT\textsubscript{a} and VIT\textsubscript{c}. We observe that it indeed realized the balanced effectiveness in suppressing the detection of STINs and in preserving the detection of stable context inconsistencies (i.e., all its values are between counterparts from VIT\textsubscript{a} and VIT\textsubscript{c}). VIT\textsubscript{b} is just one example and it represents a family of algorithm variants. It allows flexible customization, and VIT\textsubscript{a} and VIT\textsubscript{c} are its two extreme ends. Thus VIT\textsubscript{x} variants enable application developers and users to customize their STIN suppression according to actual requirements (e.g., from emphasizing the suppression of STIN detection to the preservation of stable context inconsistency detection).

From these comparisons, we observe that our three scheduling algorithms all have the ability of significantly suppressing the detection of STINs while preserving the detection of stable context inconsistencies with different trade-offs. Note that these are compared with respect to all consistency constraints in each application. Since the two applications have different constraint types, we also examine whether this affects our conclusion. Due to large combination space and result similarity, we give comparison results only for the window size of 5 but with respect to different constraint types, in Fig. 25 for the Exhibition application and in Fig. 26 for the Smart Light application.

Fig. 25 (top) and Fig. 26 (top) compare the suppression of STINs for the two applications, respectively. The IMD column data show how many STINs need to be suppressed. The right three column data show how their detection is suppressed by different scheduling algorithms with respect to different constraint types. We observe that these algorithms exhibit their corresponding aggressive, conservative and balanced nature. For example, even for extremely high STIN percentages (72.3% for Type-bud and 80.0% for Type-dsj), the VIT\textsubscript{a} algorithm can still suppress them to zero. Fig. 25 (bottom) and Fig. 26 (bottom) compare the preserving of stable context inconsistencies for the two applications, respectively. We observe similar aggressive, conservative and balanced trends. For exam-
ple, the VITc algorithm can still preserve the detection of 100% stable context inconsistencies (i.e., 0% missed stable context inconsistencies). All these can be achieved with respect to different constraint types in each application.

As a summary, we conclude our answer to research question RQ2 as follows:

*Our scheduling algorithms are effective in suppressing the detection of STINs and in preserving the detection of stable context inconsistencies. They are effective with respect to different applications, consistency constraints and vitality values. They also have different trade-offs and suit different application scenarios.*

### 5.5 RQ3: Runtime Overhead

We next evaluate the runtime overhead of our scheduling algorithms. A precise measurement of how much time is solely used by making scheduling decisions can be difficult since the scheduling and inconsistency detection are mixed in our scheduling algorithms. Therefore, we measure the total time used by each scheduling algorithm (including both scheduling and inconsistency detection). We understand that a scheduling algorithm takes time to decide whether to perform inconsistency detection for each context change with respect to each consistency constraint. On the other hand, once a scheduling algorithm decides not to perform inconsistency detection for certain context changes, it also reduces the time that would otherwise be used for inconsistency detection. Thus we are interested in whether this time reduction can kill the time used for making scheduling decisions, and its answer can be reflected by our measured total time.

We compare the total time used by the IMD algorithm and our three scheduling algorithms with different window size values in Fig. 27 (top) for the Exhibition application and in Fig. 28 (top) for the Smart Light application. The IMD line segments represent the total time used by inconsistency detection without our STIN suppression. Other line segments represent the total time of inconsistency detection when our scheduling algorithms are used. We observe that our three scheduling algorithms (VITa, VITb, and VITc) surprisingly reduced the total time to 50.0%, 54.2% and 58.3%, respectively, as compared to that of the IMD algorithm on average for the Exhibition application. For the Smart Light application, the total time was reduced to 60.2%, 73.7% and 74.6%, respectively, on average. This suggests that the time used for making scheduling decisions is very small, such that the time reduced due to reduced inconsistency detection scheduling can easily kill it and even more time is saved. Besides, we observe that VITb’s time is slightly less than those of VITa and VITc, and VITc’s time is always between those of VITa and VITc. We conjecture that this is because VITa is more
aggressive and thus reduces more inconsistency detection scheduling than \( VIT_b \) and \( VIT_c \). Since \( VIT_b \) behaves in between, its time is always between those of \( VIT_a \) and \( VIT_c \).

To check whether our conjecture is correct, we also measure the number of constraint evaluations. This number would increase when more inconsistency detection scheduling is invoked. We compare the numbers of constraint evaluations by the IMD algorithm and by our three scheduling algorithms with different window size values in Fig. 27 (top) for the Exhibition application and in Fig. 28 (top) for the Smart Light application. We observe very similar trends, i.e., relative total time exactly corresponds to relative number of constraint evaluations. Besides, we also observe that with the growth of the window size value, less constraint evaluations are made and thus the total time is also reduced. This is because with the growth of the window size value, our scheduling algorithms have increasing STINs to suppress and thus more inconsistency detection scheduling is skipped due to this suppression. More skipped scheduling leads to less constraint evaluations and thus less total time. This confirms our earlier conjecture.

As a summary, we conclude our answer to research question RQ3 as follows:

*Our scheduling algorithms have very small runtime overhead, which can even be killed by reduced inconsistency detection scheduling. As a result, the total time for handling context changes is instead reduced.*

### 5.6 RQ4: Other Comparisons

We finally compare our scheduling algorithms with several aforementioned ways of avoiding reporting STINs. Since the IMD algorithm behaves unsatisfactorily due to its immediate scheduling of inconsistency detection upon each context change, one may consider delaying a while for such scheduling. However, if each context change is delayed in the same way, it does not help suppress the detection of STINs. It only incurs unnecessary delay to applications that use the concerned contexts. Thus one may consider setting up an interval to delay the handling of a group of context changes collected during this interval. That is, every after a specified interval, one handles all context changes collected during this interval as a whole. Thus some of these context changes may be synchronized now, alleviating the asynchronous context handling problem. We name this batch-based approach BAT. We experimentally compare BAT with the IMD and other three vitality-based scheduling algorithms (i.e., \( VIT_a \), \( VIT_b \), and \( VIT_c \)). We set the delay interval to a window of size 5, corresponding to all STINs (vitality range from 1 to 5).

We compare these algorithms’ effectiveness in suppressing the detection of STINs in Fig. 29 for the Exhibition application and in Fig. 30 for the Smart Light application. The percentages in figures represent remaining STINs when different algorithms are used (with different window size values for \( VIT_a \), \( VIT_b \), and \( VIT_c \)). We observe that BAT indeed manages to suppress the detection of some STINs (as compared to the IMD algorithm), but its effectiveness is consistently worse than our three scheduling algorithms. Even compared to our conservative algorithm \( VIT_a \), BAT still behaves worse, e.g., 83.3–267.4% worse for the Exhibition application and 582.7–2,690.2% worse for the Smart Light application. This is understandable since setting up such a delay does not fully address the asynchronous context handling problem, not to mention that STINs are not only caused by asynchronous context handling, as we discussed earlier.

BAT also delays the handling of context changes, and the delay is on average a window of size 2.5. For the Exhibition application, this delay is about 5 seconds, and for the Smart Light application, the delay is about 20 seconds (the two applications have different window lengths). Such large delays can be undesirable to context-aware applications that are supposed to react to environmental changes in time. For example, consider that a light’s illumination starts to be adjusted after 20 seconds, which can be long enough for a worker to already leave this illumination range. On the other hand, our three scheduling algorithms try to minimize this delay by immediately invoking or skipping inconsistency detection once the scheduling decision is made for collected context changes. As such, they incur much less delay. For the Exhibition application, we measured the delay to be a window of size 0.27, 0.50, 0.69, 0.85 and 0.99, respectively, when our scheduling algorithms’ window size value is set to 1, 2, 3, 4 and 5. For the Smart Light application, the delay incurred by our scheduling algorithms is a window of size 0.17, 0.28, 0.35, 0.41 and 0.46, respectively, which is even less. Then even if our scheduling algorithms adopt its largest window size of 5 for suppressing the detection of all STINs, they would incur only a delay of 2 seconds and 3.7 seconds for the two applications, respectively, as shown in Table 4 and Table 5. As a comparison, BAT incurs a much longer delay (150% and 441% longer for the two applications, respectively), not to mention its low effectiveness in suppressing the detection of STINs.

One may also consider first detecting all context inconsistencies and then waiting for a while to see whether
some of them will vanish automatically, as we discussed earlier. We name this approach \textsc{Wat}. While \textsc{Wat} can help identify STINs, it incurs even large delay. The delay has to be up to a window of size 5, as one has no idea whether any STIN of any vitality exists in detected context inconsistencies. For the Exhibition and Smart Light applications, this delay is about 10 seconds and 40 seconds, respectively, as shown in Table 4 and Table 5. Compared to delays incurred by our scheduling algorithms, they are 400\% and 981\% longer, which are clearly too large.

Besides large delay, \textsc{Wat} has to detect all context inconsistencies and then examine them for STINs. This costs more time than necessary due to unnecessary inconsistency detection, e.g., 71.4–100\% more time for the Exhibition application and 34.1–66.0\% more time for the Smart Light application according to Fig. 27 and Fig. 28 (using the IMD data as they are the same as the \textsc{Wat} data). For small-scale Exhibition and Smart Light applications, this might be acceptable. However, for large-scale applications that are undergoing continual and frequent context changes, this can cause unexpected consequences due to large workload caused by such extra inconsistency detection. We in the following subsection present a large-scale case study with our third application (Smart City) to study the effectiveness of different scheduling algorithms in practice.

As a summary, we conclude our answer to research question RQ4 as follows:

\textit{Our scheduling algorithms are more effective in suppressing STINs than batch-based scheduling. Besides, our algorithms incur much less delay than batch-based or wait-based scheduling.}

### 5.7 Case Study

We in this subsection present a real-world application scenario and investigate different scheduling algorithms' effectiveness in suppressing the detection of STINs in practice. This scenario concerns a city in South China, which endeavors for its smart city perspective [56], [61]. This Smart City application focuses on the city's traffic conditions for hot areas as well as the whole city. The hot areas are of particular interest, and their traffic conditions can affect how the city's smart city perspective is realized. The application maintains for each hot area a collection of recent taxi conditions (e.g., GPS location, driving speed, driving direction and service status) for taxis driving in this area. This collection of taxi conditions can be regarded as a dynamic context associated with each hot area. Based on them, the application analyzes traffic conditions for this area as well as the whole city, and studies how they evolve with time. By doing so, the application supports smart activities such as dynamic routing, i.e., suggesting the most efficient route to a destination for a driver and dynamically adjusting it when traffic conditions along this route deteriorate with time [56], [61].

We monitored 760 taxis from one company for continuous 24 hours, and collected about 1.55 million taxi data. These data correspond to a total of 6.69 million context changes to the application (each taxi datum can trigger more than one context change). The interval between every two context changes is 12.9 ms on average, varying from 4.6 ms to 692.7 ms. These taxi data impose concrete challenges on their effective and efficient handling. Engineers found many problems with the application's contexts, e.g., some taxi data suggest taxis driving out of the city's scope or even in sea (the city is near sea), driving in a unreasonably high speed like over 200 km per hour (impossible for this crowded city), driving with unreasonable location jumps (from one place suddenly to another), belonging to different hot areas that are far away from each other at the same time, or disappearing from a hot area although actually driving there. These indicate realistic noises in the taxi data. Application engineers designed assertion checkers to detect such problems in contexts, and handled them before use by the application. We expressed these assertions into 16 consistency constraints. As mentioned earlier in Table 3, our \textsc{Cina} tool derived 61 instability conditions from these constraints. We use these instability conditions in our scheduling algorithms, and compare them to the IMD algorithm about their effectiveness in suppressing the detection of STINs. The comparison was conducted for the 6.69 million context changes that were faithfully fed to the application according to their actual timestamps.

In previous subsections (5.2–5.6), we have compared different scheduling algorithms in suppressing the detection of STINs by controlled experiments. In this subsection, we focus on a case study setting by fixing some parameters according to application requirements. We selected the \texttt{VIt}, algorithm as it aims for best preserving the detection of stable context inconsistencies and this is also the requirement from engineers in designing assertions for the Smart City application. We selected the IMD algorithm for comparison as it represents the existing inconsistency detection practice. Besides, we also considered an additional factor, inconsistency detection technique. Different techniques have different efficiency levels, and high efficiency is critical to handling frequent context changes. This is because if the handling is not in time, extra STINs (false positives) and missed stable context inconsistencies (false negatives) can result [61], and this would affect our measurement of these two metrics. We thus considered two inconsistency detection techniques, namely, \texttt{ECC} and \texttt{PCC} [61]. \texttt{ECC} is a traditional incremental checking technique that rechecks a consistency constraint only when a context change affects it. \texttt{PCC} is our previous

### Table 4. Delay Comparisons for the Exhibition Application (Delay Measured by Window Size and Time).

<table>
<thead>
<tr>
<th>Delay</th>
<th>\textsc{Bat}</th>
<th>\textsc{Wat}</th>
<th>\texttt{VIt}/\texttt{VIt}/\texttt{VIt}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Window size (Time (s))</td>
<td>2.5</td>
<td>5</td>
<td>0.27 0.50 0.69 0.85 0.99</td>
</tr>
<tr>
<td>Time (s)</td>
<td>5</td>
<td>10</td>
<td>0.5 1 1.4 1.7 2</td>
</tr>
</tbody>
</table>

### Table 5. Delay Comparisons for the Smart Light Application (Delay Measured by Window Size and Time).

<table>
<thead>
<tr>
<th>Delay</th>
<th>\textsc{Bat}</th>
<th>\textsc{Wat}</th>
<th>\texttt{VIt}/\texttt{VIt}/\texttt{VIt}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Window size (Time (s))</td>
<td>2.5</td>
<td>5</td>
<td>0.17 0.28 0.35 0.41 0.46</td>
</tr>
<tr>
<td>Time (s)</td>
<td>20</td>
<td>40</td>
<td>1.4 2.2 2.8 3.3 3.7</td>
</tr>
</tbody>
</table>
work on partially rechecking every constraint that has to be rechecked. Details of the two techniques are not our focus in this article, but we note that PCC can work much faster than ECC (28.6 times faster for the Smart City application). Thus we measured and compared the effectiveness of different scheduling algorithms in suppressing the detection of STINs, and investigated how it can be affected by different inconsistency detection techniques.

With the **PCC technique**. We used the same hardware and software configurations as introduced earlier. We first used the PCC technique to alleviate the efficiency concern, since our focus is the effectiveness of different scheduling algorithms. Regarding the 6.69 million context changes fed to the Smart City application, the IM algorithm detected a total of 199,250 context inconsistencies. Unfortunately, a huge portion of them, 92.8% (184,848), turns out to be STINs. Further study discloses that these STINs include vitality-1 and vitality-2 ones, which occupy 80.2% (148,221) and 19.8% (36,627), respectively. As a comparison, our VT algorithm (window size set to 2) detected a total of 14,422 context inconsistencies, among which the STIN percentage is 0.1% (only 20). The STIN suppression is nearly 100% (from 184,848 to 20), which is significant. Besides the STIN suppression, we also measured missed stable context inconsistencies. We analyzed all context changes against 16 consistency constraints in an offline way and identified a total of 14,402 stable context inconsistencies (as oracle data). We compared detected context inconsistencies by the two scheduling algorithms to the oracle data to calculate missed stable context inconsistencies. We observe that neither the IM algorithm nor our VT algorithm missed any stable context inconsistency (i.e., both zero). We owe this achievement to the PCC technique, as further study discloses that both algorithms handled 100% context changes and no inconsistency was missed. We thus conjecture that when inconsistency detection can work efficiently enough, scheduling algorithms can solely decide the quality of detected context inconsistencies (i.e., whether to contain STINs and how many they could be). Table 6 lists these comparison data when the PCC technique is used.

**With the **ECC** technique. We then used the ECC technique to handle the 6.69 million context changes with the IM and VT algorithms, respectively. This time the IM algorithm detected a total of 181,675 context inconsistencies, which is slightly less than that (199,250) when using the PCC technique. However, the STIN percentage is still very high: 93.0% (169,016), very close to that (92.8%) when using the PCC technique. As a comparison, the VT algorithm detected a total of 26,647 context inconsistencies, which is much more than that (14,422) when using the PCC technique. Besides, the STIN percentage is also greatly increased to 50.5% (13,458) from that (0.1%) when using the PCC technique. Further study discloses that using the ECC technique led to low checking efficiency and some context changes were not handled in time or even lost. Then STINs caused by these context changes were no longer detectable. Thus the IM algorithm detected less STINs (15,832 less). However, for the VT, algorithm, since it previously detected only 20 STINs when using the PCC technique, these delayed or lost context changes almost did not help much in reducing previous STINs, but instead caused wrong inconsistency detection due to misalignment or loss of context changes. That is why the VT algorithm detected new STINs (or more strictly, they are new false positives, not related to earlier ones detected by the PCC technique). Even so, we observe that the VT algorithm detected much less STINs than the IM algorithm (155,558 less or 92.0% less) with the ECC technique. This shows our VT algorithm’s effectiveness in suppressing the detection of STINs even when the inconsistency detection is of low efficiency. Table 6 lists these comparison data when the ECC technique is used.

We also measured missed stable context inconsistencies when using the ECC technique. The IM algorithm missed 1,743 stable context inconsistencies, which occupy 12.1% of the aforementioned 14,402 oracle data. This rate can be up to 31.6% for certain rush hours (e.g., 6pm). Our VT algorithm missed 1,213 stable context inconsistencies, whose percentage is 8.4%. Since both IM and VT algorithms did not miss any stable context inconsistency when using the PCC technique, these missed stable inconsistencies should be caused by using the ECC technique, which worked much slower than the PCC technique. This is because when inconsistency detection was of low efficiency, some context changes were not handled in time or even lost. This caused inconsistency detection to fail to report stable context inconsistencies related to these context changes. Even so, we observe that our VT algorithm missed less stable context inconsistencies than the IM algorithm (530 less or 30.4% less). This is because our VT algorithm reduces unnecessary inconsistency detection scheduling, thus alleviating the efficiency problem caused by the ECC technique. We observe that this difference helped reduce missed stable context inconsistencies, thus validating our VT algorithm’s additional advantage on this aspect, besides its main effectiveness in suppressing the detection of STINs. Table 6 also lists these comparison data when the ECC technique is used.

As a summary, we conclude our results in this case study as follows:

**Our vitality-based STIN suppression works in practice.** For our large-scale Smart City study, our work reduced STINs by nearly 100% when using efficient PCC incon-

---

**TABLE 6.** **Detected STINs and missed stable context inconsistencies comparison for the Smart City application.**

<table>
<thead>
<tr>
<th></th>
<th>PCC</th>
<th>ECC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># detected</td>
<td># STINs</td>
</tr>
<tr>
<td></td>
<td>inconsistencies</td>
<td>(92.8%)</td>
</tr>
<tr>
<td></td>
<td>199,250</td>
<td>184,848</td>
</tr>
<tr>
<td></td>
<td>14,422</td>
<td>184,828 (&lt;100%)</td>
</tr>
<tr>
<td>Reduction (VT, vs. IMD)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
sistency detection, and by 92.0\% when using inefficient Ecc inconsistency detection. It also largely reduced missed stable context inconsistencies (by 30.4\%), which were caused by inefficient Ecc.

5.8 Threats to Validity
The largest threat may relate to the external validity of our experiments and conclusions. Our experiments selected only three applications, which may not be fully representative of all context-aware applications. We alleviate this threat by diversity and completeness of consistency constraints used in these applications. First, these applications exhibited wide ranges of STIN percentages (32.9–93.0\%), which indicate their diverse nature. Second, different types of consistency constraints exhibit diverse subjection to the detection of STINs (25.8–80.0\%), even within the same application. Third, these consistency constraints cover all seven formula types in the constraint language. Fourth, we conducted real-world experiments for studying how STINs are and how they can be suppressed in practice, and these experiments handled up to millions of context changes. We believe that all these efforts can help alleviate as much as possible the threat that might affect the external validity of our experiments and conclusions.

There are also confounding factors that may internally affect our measurement of detected STINs and missed stable context inconsistencies. For example, we fixed hardware and software configurations in our experiments. Configuration changes can lead to performance difference in performing inconsistency detection. We do have considered this. Our first two applications are small-scale and thus the performance concern is not demanding. For our third application, its context changes are continual and frequent. We considered two representative inconsistency detection techniques (Pcc and Ecc). Since their performance difference is large (about 28.6 times), they served as two extreme ends for studying whether the inconsistency detection efficiency would affect the effectiveness of scheduling algorithms. We observed that the inconsistency detection efficiency does affect our measurement of detected STINs and missed stable context inconsistencies. However, it does not change our conclusion that our vitality-based scheduling has an overwhelming advantage over immediate scheduling (nearly 100% and 92.0\% STIN reduction under Pcc and Ecc, respectively). Besides, we adjusted the sliding-window size in our Vt algorithms to cover from minimal STINs to all STINs. We observed consistent trends that our Vt algorithms greatly suppressed or even completely avoided the detection of STINs. Due to these efforts, we can safely conclude that those confounding factors, if any, would not significantly change our measurement data and can still reach the same conclusions.

We also considered construct validity. While STIN suppression is our goal, missed stable context inconsistencies are also a key metric that decides the usefulness of our STIN suppression. For example, simply skipping inconsistency detection can easily realize zero STIN, but it is clearly useless as all stable context inconsistencies are also lost. Therefore, we measured both metrics for a fair comparison. Experiments with the first two applications disclosed different effectiveness and trade-offs of our vitality-based scheduling algorithms. We thus suggested different application scenarios for different algorithms. Experiments with the third application disclosed that our scheduling algorithms worked even more satisfactorily when deployed in large-scale application scenarios in which context changes are continual and frequent. Besides, we measured the delay caused by different scheduling algorithms, which may be the concern of applications that care about responsiveness. Some algorithms have to delay the handling of context changes, some delay the reporting of detected context inconsistencies, and some incur internal delay when deciding the scheduling of inconsistency detection. We unified all these as the delay between a context change is collected and a corresponding context inconsistency is reported. We observed that our scheduling algorithms also have nice performance on this aspect by incurring much smaller delay. We expect that all these efforts can contribute to a fair comparison of different scheduling algorithms in their effectiveness and trade-offs.

Finally, regarding theoretical reliability, we tried to make our experiments less biased and conclusions more reliable. For example, we made different scheduling algorithms share the same context and change data structures in implementation. They differ only in how to schedule inconsistency detection. All consistency constraints used in our experiments can be released for reference. Our CINA tool can be made publically available for other users to derive instability conditions for their own consistency constraints. By these efforts, we aim for more theoretical reliability and wider application scenarios in which our work can practically improve the quality of context inconsistency detection results.

6 Related Work
In this section, we present and discuss related work on five aspects. They are: context modeling for pervasive computing, dependability assurance for context-aware applications, consistency management for software artifacts, tolerance for detected inconsistencies, and hazard or predicate detection for distributed events. These five aspects closely relate to our context inconsistency detection and STIN suppression work in this article.

Context modeling for pervasive computing. In recent years, pervasive computing and context-awareness keep receiving increasing attention. Various applications and supporting middleware infrastructures are being proposed and developed. Many applications, like ConChat [43] and ActiveCampus [18], and middleware infrastructures, like CARISMA [9], LI.ME [37], EgoSpaces [27], Gaia [42] and Cabot [58], support interesting context-aware properties. These properties enhance applications by supporting their runtime switching of different working modes or functionalities, thus delivering smart services to suit varying user requirements in an autonomous way.

Context-aware applications use various models to rep-
resent and manipulate their contexts. These include early key-value and markup models such as CC/PP [30], domain-focused models such as the W4 context model [10], later fact-based models [20], [58], ontology-based models such as the CARE framework [2], and hybrid models [5]. In this article, we do not aim to propose new context models. Instead, we use key-value models for representativeness as they are widely used in applications or supporting other models. We show that even with such simple key-value models, applications can still be subject to unstable inconsistency detection results when key-value pairs are being added, deleted or updated in an asynchronous or separate way. Therefore, our observations and discussions made in this article can also impact other context models as well.

Besides, context fusion [1] may be used when there is a need for deriving high-level information from raw sensory data (i.e., sensor fusion [4], [35]) or from multiple pieces of low-level information for complicated reasoning tasks (e.g., for human activity recognition [8]). Context fusion may probably be first mentioned in Context Toolkit [15], [48], where a set of context widgets provide applications with access to context information while hiding the details of context sensing and fusion. Later, Chen et al. [12] developed a larger context fusion network, Solar, for software reusability and heterogeneity hiding. Context fusion relates to context consistency management in that the later can be one of the first’s functionalities logically. However, in most cases they have different focuses. Context fusion is for integrating information from multiple sensors and deriving new information. Its focus is the derivation specification. Our focus in this article is that even if the specification is correct, the derived information may still be wrong if the derivation is conducted at inappropriate time. Therefore, our work complements context fusion work for its quality.

**Dependability assurance for context-aware applications.** While the benefits of context-aware applications are plenty, their development has shown to be more challenging than traditional applications by exhibiting more error-prone nature [33], [46], [47], [62], [63].

Errors in context-aware applications mostly relate to low-quality contexts and their improper handling in applications. Contexts are derived from sensory data, which are continually collected from ever-changing environments. One outstanding challenge is the imperfect nature of contexts due to unpredictable and uncontrollable noises arising from environments [34], [44], [58]. Since such noisy contexts are always the fact, one line of work focuses on identifying defects in applications that have failed to consider context handling adequately or correctly. For example, Wang et al. [52] studied thread switching at context-aware program points in applications, and proposed strengthening test coverage for such switching in order to identify more context related defects. Lu et al. [34] studied new test coverage criteria for context-aware applications that run with context management services (e.g., context inconsistency detection and resolution as discussed in this article). Such applications may incur new or modify existing data flows, and defects associated with such data flows can be hard to expose by traditional testing techniques. Sama et al. [46], [47] and our previous work [33], [62] studied defects in model-based context-aware applications, which can be triggered by unanticipated context changes and lead to undeterministic or unstable adaptations. These pieces of work tried static analysis or dynamic tracking to expose context related defects in applications. They examine whether such applications have been designed properly to handle noisy contexts as required.

Another line of work more relates to our context handling work in this article, by controlling the negative impact of noisy contexts to applications [44], [58]. This is done by transforming such noisy contexts to new ones that meet certain quality requirements from users or applications. Some pieces of work proposed filter or threshold based techniques to smooth noisy contexts [26], or measured them probabilistically with different uncertainty levels for reference by applications [29]. However, due to the lack of a precise oracle for generally validating different types of contexts and their changes [61], these techniques are either application-specific (e.g., relying on application domain knowledge) or context-specific (e.g., focused on RFID contexts only). Besides, they focus on data-level errors (e.g., recognizing abnormal values) and cannot prevent context inconsistencies (which typically concern different types of contexts with application semantics) at an application level. Therefore, there still exists a need for comprehensive context consistency management for applications.

**Consistency management for software artifacts.** Consistency management for traditional software artifacts has been extensively studied in our communities for many years. Typical examples of such software artifacts include UML models [6], [16], XML files [38], [40], [45], configurations [55], and data structures [14]. Inconsistencies detected from these software artifacts against prespecified consistency constraints are usually presented to users for inspection. This leads to interactive repairing [17], [39], or manual or semi-automated resolution based on predefined strategies [11], [14], [54] or heuristic rules [13], [60]. These well studied software artifacts are usually static or change rarely or slowly.

On the other hand, contexts studied in this article are subject to continual and frequent changes in pervasive computing. They are dynamic software artifacts. This implies two expected efforts in checking them for inconsistency. First, context inconsistency detection should be made efficiently. Our previous work [61] aimed at this aspect by proposing a partial constraint checking technique, which can work much faster than traditional incremental checking techniques. Later we extended it to support CPU-level [56] and GPU-level [49] parallel checking. Second, the inconsistency detection should be made effectively so as to report proper context inconsistency results. This concern is important because there is no clear criterion of judging boundaries in continual context changes for scheduling proper inconsistency detection. We note that this may not be a problem for inconsistency detection of traditional software artifacts, which are static.
or change rarely or slowly. It can be easy to decide their changing boundaries (e.g., after committing a series of performed model changes). This effective inconsistency detection concern is the focus of this article.

We address this concern by automatically recognizing key pairs of context changes against derived instability conditions and selectively scheduling inconsistency detection to suppress the detection of STINs. Previously, we tried to address this by learning such conditions from history inconsistency detection data [53], but its effectiveness depends on how representative the training data are and it can never achieve 100% STIN suppression as realized by our constraint instability analysis and dynamic inconsistency detection scheduling in this article. This studied effectiveness concern considers inconsistency detection from a new perspective, and it complements the first efficiency concern together towards the quality of inconsistency detection results.

**Tolerance for detected inconsistencies.** Our STIN suppression work resembles the inconsistency tolerance idea as both advocate not handling some detected inconsistencies. While there are many existing pieces of work on inconsistency tolerance, we discuss some representative and state-of-the-art ones for example and explain key differences.

Nuseibeh et al. [41] proposed tolerating inconsistencies in software development and discussed benefits of doing so in certain scenarios. Our work echoes this point: some context inconsistencies should not be handled since they are unstable and can vanish by themselves, and handling them can instead cause system instability. The difference lies in that the inconsistency tolerance work detects inconsistencies first and then decides how to deal with them, while our work makes STINs undetected and saves corresponding computational resources. Besides, inconsistency tolerance may take time as it depends on the impact and risk associated with inconsistencies that are continuously evaluated [41]. For our problem, context-aware applications are expected to respond to context changes sensitively. This may not be satisfied by delayed decisions on whether detected inconsistencies should be handled or not.

The most recent inconsistency tolerance work can be Kehrer et al.’s edit script generation work [28]. It reports that wrong grouping of low-level model changes may produce inconsistent states in model evolution and such inconsistent states may not need handling. However, this work focuses on a different problem from ours: Given two model versions that are consistent, how to make each evolution step between them also consistent? Our work in this article does not assume contexts to be inconsistency-free. Instead, it aims to detect stable inconsistencies upon context changes and at the same time suppress the detection of STINs. This can be more complicated. Besides, the edit script generation work is in an offline manner. It assumes the availability of two model versions and all mode changes between them, and then figures out how to generate consistent intermediate steps. Instead, our work is in an online manner. It does not have the knowledge of future context changes. It needs to predict whether current context changes would incur stable inconsistencies or STINs with future changes.

**Hazard or predicate detection for distributed events.** Some occurrences of STINs can be due to asynchronous context changes, which cause misalignment of sensory data. This phenomenon is not unique to context consistency management. It can be similar to signal hazards in sequential digital circuits [51] or context hazards in model-based context-aware applications [46]. In these application scenarios, misaligned signals or context combinations can be encountered and cause unanticipated functionality triggering. Nevertheless, we note that these issues have different scopes and challenges from our focus in this article. Existing work assumes logical expressions under evaluation to be specified by propositional logic and Boolean (true or false) value changes. Our work focuses on more complex consistency constraints, which can be specified by first-order logic and arbitrary (any-valued) context changes. Therefore, our work extends existing techniques and complements them for wider application scenarios.

Our work also relates to predicate detection in distributed environments [21], [31], [32], [64] in the sense that distributed events or context changes are collected and predicates or constraints are evaluated upon these events or context changes. Nevertheless, we note two distinct differences. First, the consistency constraints discussed in this article are built on complex predicates and their logical combinations (e.g., logical formulas can be deeply nested and arbitrarily guarded with universal or existential operators). Distributed evaluation of such constraints has not been supported by these pieces of existing work. Second, we focus on analyzing a constraint’s syntactic structure and built-in semantics, and examining whether it is subject to the detection of STINs. Later we use derived instability conditions to proactively suppress the detection of STINs. These pieces of existing work, however, care for communication protocols that broadcast distributed events and compute predicate values. Therefore, we have different focuses. Still, we may consider in future more scenarios in which these techniques may cooperate towards the quality of predicate detection results by efforts in both event communication and constraint evaluation.

7 Conclusion
In this article, we studied the STIN issue in pervasive computing. We observed that STINs commonly exist in practice and negatively affect context-aware applications, but this problem has not received adequate attention or been systematically studied. In pervasive computing, context changes are generated in a continual and infinite way, and their collection is subject to natural asynchronism. Besides, different applications have different semantics in deciding boundaries of related context changes, but this information is not generally available. It thus becomes difficult to perform inconsistency detection at appropriate time, leading to the detection of STINs. We showed the significance of this problem, and explained why tradi-
tional immediate scheduling of inconsistency detection incurs numerous STINs. To address this issue, we presented a constraint instability analysis tool, CINA, to automatically derive instability conditions from consistency constraints. These conditions can be used to systematically suppress the detection of STINs, while still preserving the detection of stable context inconsistencies. When combined with different strategies, our STIN suppression can have different effectiveness and trade-offs, thus catering for different application requirements. A distinct feature of our STIN suppression is its total transparency to application developers or users as it works in an application-independent way.

We evaluated our work on real-world context-aware applications with both small-scale field tests and large-scale city-wide tests. The experiments reported promising results. We plan to further validate our work in more application scenarios, e.g., for smartphone applications, which are equipped with various sensing devices of different sampling cycles [46]. Besides, we may explore the balance between suppressing the detection of STINs and preserving the detection of stable context inconsistencies, and study ways of dynamic trade-off estimation and balance tuning. Finally, in this work we used a window parameter by a vitality value, which can also be adapted to a time interval. In practice, people may observe the detection of STINs at application runs and use the detected STINs’ maximal duration as the vitality value [53]. This is an application-specific choice and what we provide is customizable STIN suppression with varying effectiveness and trade-off. In this article, all studied STIN suppression approaches rely on this vitality value to decide how long one should wait to avoid reporting STINs, and our proposed approach has achieved the shortest delay as the evaluation shows. Still, deciding the exact vitality value requires application runs or human experience, which might need additional effort. In future, we are interested in exploring systematic ways of adapting the window size internally at application runs without the vitality input from users. We are working along these lines.

**ACKNOWLEDGMENT**

The authors wish to thank Sam Malek and anonymous reviewers for their valuable comments on improving this article. 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, and by Research Grants Council (611813) of Hong Kong. All correspondence should be addressed to Chang Xu (email: changxu@nju.edu.cn, tel: +86-89680919, fax: +86-83593283).

**REFERENCES**


Chang Xu received his doctoral degree in Computer Science and Engineering from The Hong Kong University of Science and Technology. In 2010, he joined Nanjing University, where he is an associate professor with the State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology. He participates actively in program and organizing committees of major international software engineering conferences. He co-chaired the FSE 2014 SEES Symposium and MIDDLEWARE 2013 Doctoral Symposium. His research interests include software engineering, software testing and analysis, and pervasive computing.

Wang Xi is an MSc student with the Department of Computer Science and Technology, Nanjing University. He received his BSc degree in Computer Science and Technology from Nanjing University. His research interests include software engineering and pervasive computing.

S.C. Cheung received his doctoral degree in Computing from the Imperial College London. In 1994, he joined The Hong Kong University of Science and Technology, where he is a full professor of Computer Science and Engineering. He participates actively in program and organizing committees of major international software engineering conferences. He was the General Chair of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014). He is the North-Asia Liaison in the ACM SIGSOFT executive committee. He is a director of the Hong Kong R & D Center for Logistics & Supply Chain Management Enabling Technologies. His research interests include program analysis, testing and debugging, big data software, cloud computing, internet of things, and mining software repository.

Xiaoxing Ma received his PhD degree in Computer Science and Technology from Nanjing University. He is a professor at the Department of Computer Science and Technology at Nanjing University. He also serves as the Deputy Director of the Institute of Computer Software at the same university. His research interests include self-adaptive software systems, software architecture and middleware systems.

Chun Cao received his PhD degree in Computer Science and Technology from Nanjing University. He is an associate professor at the Department of Computer Science and Technology at Nanjing University. His research interests include software architecture, middleware systems and cloud computing.

Jian Lu received his doctoral degree in Computer Science and Technology from Nanjing University. He is currently a full professor with the Department of Computer Science and Technology and the Director of the State Key Laboratory for Novel Software Technology at Nanjing University. He has served as a Vice Chairman of the China Computer Federation since 2011. His research interests include software methodologies, automated software engineering, software agents, and middleware systems.