GEAS: Generic Adaptive Scheduling for High-efficiency Context Inconsistency Detection

Bingying Guo[†‡], Huiyan Wang[†‡], Chang Xu[*†‡], Jian Lu[†‡]
[†]State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
[‡]Department of Computer Science and Technology, Nanjing University, Nanjing, China
bingying_nju@outlook.com, cocowhy1013@gmail.com, {changxu, lj}@nju.edu.cn

***Abstract*—Context-aware applications adapt their behavior based on collected contexts. However, contexts can be inaccurate due to sensing noise, which might cause applications to misbehave. One promising approach is to check contexts against consistency constraints at runtime, so as to detect context inconsistencies for applications and resolve them in time. The checking is typically immediate upon each collected context change. Such a scheduling strategy is intuitive for avoiding missing context inconsistencies in the detection, but may cause low-efficiency problems for heavy-workload checking scenarios, even if equipped with existing incremental or parallel constraint checking techniques. One may choose to check contexts in a batch way to increase the efficiency by reducing the number of constraint checking. However, this can easily cause missed context inconsistencies, denying the purpose of inconsistency detection. In this paper, we propose a novel scheduling strategy GEAS of two nice properties: (1) adaptively tuning the batch window to avoid missing any context inconsistency; (2) generic to checking techniques with no or little adjustment. We experimentally evaluated GEAS against the immediate strategy with existing constraint checking techniques. The experimental results show that GEAS achieved 143–645% efficiency improvement without missing any context inconsistency, while alternatives caused 39.2–65.3% loss of detected context inconsistencies.**

***Index Terms*—Context inconsistency, scheduling strategy, suspicious pair**

I. Introduction

Consistency management for software artifacts (e.g., XML documents [1], UML models [2], data structures [3], ...) has been extensively studied for their maintenance during software development and evolution. *Contexts*, key software artifacts for supporting smart adaptation, play an important role in context-aware applications. Contexts also require consistency management to avoid applications to behave abnormally due to sensing noise, leading to corrupted contexts [4]. Typical contexts, like user location, room temperature and GPS data, are subject to frequent changes, and thus call for efficient and effective consistency management [5]–[7]. This is typically done by specifying necessary properties that must hold for contexts as *consistency constraints* [1], [8] and checking contexts against the constraints to see whether any violation (named *context inconsistency* [6], [9]) occurs. Detected context inconsistencies can then be resolved by context fixing to help applications to behave normally and smartly.

*Corresponding author.

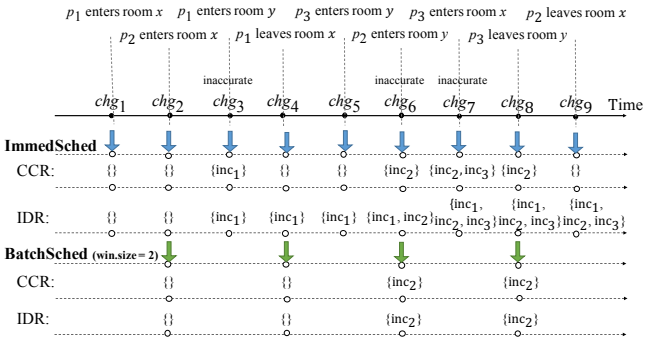

Fig. 1: Illustrative scenario – problem with BatchSched (CCR: constraint checking result; IDR: inconsistency detection result).

Typically, checking contexts against consistency constraints is scheduled *immediately* upon each collected *context change* (i.e., any change relating to a specific context) to detect potential inconsistencies. However, it can cause low-efficiency problems when it comes to heavy-workload checking scenarios, even if equipped with incremental [6] or parallel [7] constraint checking techniques. Unfortunately, heavy-workload scenarios are common in practice, since real-life context changes usually come in a stream and very frequently. As a result, immediate scheduling could be impractical for application under many circumstances. For ease of presentation, we name such checking strategy upon each collected context change *Immediate Scheduling* (*ImmedSched*).

In order to improve the efficiency of context inconsistency detection, one may choose to check contexts in a batch way by reducing the number of constraint checking, which we name *Batch Scheduling* (*BatchSched*). BatchSched checks contexts upon multiple collected context changes as a group. It clearly works faster than ImmedSched, since it merges the checking for multiple context changes into one and some redundant overhead can thus be saved. However, such a scheduling strategy can easily cause undesirable consequences like missing context inconsistencies, causing applications still to behave abnormally.

To see it, we consider an application scenario as illustrated in Fig. 1. The application's contexts specify the persons currently staying in each room (rooms x and y). Suppose that there are nine context changes collected in a sequence, and that changes chg_3, chg_6 and chg_7 are inaccurate (e.g.,

wrongly collected due to sensing noise or collected in wrong time due to sensing asynchrony [10], [11]). We consider a consistency constraint for this scenario: *no one can stay in rooms x and y at the same time*. Then we observe that context changes chg_3, chg_6 and chg_7 cause three context inconsistencies (inc.), respectively, with earlier context changes. In the figure, CCR represents constraint checking result, i.e., inconsistencies for the current contexts with all changes so far applied, and IDR represents inconsistency detection result, i.e., all inconsistencies ever detected so far. If one applies ImmedSched, constraint checking would be scheduled upon each context change, as indicated by blue arrows in Fig. 1. Then we observe that ImmedSched perfectly detects all context inconsistencies (inc_1, inc_2, inc_3), although it is rather time-consuming.

However, if one applies BatchSched and checks these context changes in a batch of size two (i.e., window size is two), constraint checking would be scheduled every two changes (i.e., upon chg_2, chg_4, ..., chg_8), as indicated by green arrows in Fig. 1. Although CCR still reports the same results (relying on current contexts only), IDR can have quite different results (accumulating all inconsistencies ever detected). For example, since constraint checking is not scheduled upon context changes chg_3 and chg_7, their incurred inconsistencies (inc_1, inc_3) are thus missed. As a result, the final IDR for BatchSched contains inc_2 only, accounting for a missing rate of 66.7%, seriously impairing the purpose of context inconsistency detection.

Such inconsistency-missing consequences are undesirable and can still cause context-aware applications to misbehave, since their adaptive behavior relies on contexts, which are still inconsistent. Besides, even if equipped with state-of-the-art inconsistency resolution [4], [12] or exception handling services [13], [14], the applications can seldom survive since inconsistencies have been already missed.

Therefore, we face a dilemma situation: (1) ImmedSched does not miss context inconsistencies, but it is too slow; (2) BatchSched works faster, but it misses context inconsistencies in the detection. To address this dilemma problem, we in this paper propose a novel scheduling strategy *GEneric Adaptive Scheduling* (*GEAS*). GEAS can both improve the efficiency of context inconsistency detection by batch-styled constraint checking and avoid missing any context inconsistency by adaptive tuning of the batch window size. GEAS owns two nice properties: (1) the adaptive tuning of the window size is fully automated; (2) the scheduling is generic to many existing constraint checking techniques (e.g., ECC [1], PCC [6], Con-C [7] and GAIN [5]) with no or little adjustment. The key insight behind GEAS is that when checking context changes in a batch, only certain combinations of context changes in a batch can cause context inconsistencies missed in the detection. One can derive such combinations from consistency constraints in advance (named *suspicious pairs*). One then uses such suspicious pairs at runtime to judge whether any context change in a stream can potentially cause context inconsistencies missed before checking the change in a batch way. This automated judgment helps adaptively tune the batch window size, such that the efficiency of inconsistency detection can be smartly improved without compromising its effectiveness, i.e., missing any inconsistency.

We conducted experiments to evaluate GEAS' efficiency and effectiveness. Our experimental results report that GEAS achieved 143–645% efficiency improvement when equipped with various constraint checking techniques, as compared with ImmedSched and BatchSched. The results also report that GEAS completely avoided missing any context inconsistency in the detection, while alternatives caused 39.2–65.3% loss of context inconsistencies in the detection.

In summary, we make the following contributions in this paper:

- We propose a novel scheduling strategy GEAS for context inconsistency detection. GEAS can adaptively tune its batch window to both improve the detection efficiency and avoid missing context inconsistencies.
- We elaborate on GEAS' generality and explain how to apply it to four state-of-the-art context constraint checking techniques with no or little adjustment.
- We compare GEAS' efficiency and effectiveness with existing work and evaluate its practical performance in real-world scenarios.

The remainder of this paper is organized as follows. Section II introduces our problem background and formulation. Section III elaborates on our GEAS approach. Section IV explains the application of our GEAS to existing context constraint checking techniques. Section V evaluates GEAS and compares its efficiency and effectiveness to existing work. Finally, Section VI discusses the related work in recent years, and Section VII concludes this paper.

II. Background and Problem Formulation

In this section, we introduce the background to context constraint checking with some concepts defined, and then present our problem formulation.

A. Concepts

A *context* refers to a piece of environmental or logical information interesting to a context-aware application [10]. We model a context as a finite set of specific elements, and each element represents a relevant part of this context. Take a context-aware application App_room as an example, which can automatically turn on a room's air conditioning system if the room is occupied with many people. We model all people in room x at the moment by a context $R_x = \{p_1, p_2, ...\}$. R_x represents the set of all people currently in room x and p_i identifies each individual person.

A *context change* denotes any possible change relating to a specific context, and it is represented by a tuple $<type, context, para1, para2, \ldots>$. There are three types of context changes, namely, *addition change* (adding a new element into a context), *deletion change* (deleting an existing element from a context) and *update change* (updating an element's value). For ease of presentation, we use "+", "−" and "#" to represent the

three types, respectively. Suppose that a context for application App_room is R_x = $\{p_1, p_2\}$ (i.e., only two persons p_1 and p_2 are currently in room x). An addition change (e.g., p_3 enters room x) can be represented as $<+, R_x, p_3>$, and a deletion change (e.g., p_2 leaves room x) as $<-, R_x, p_2>$. An update change (e.g., p_3 updates his personal information) can be represented as $<\#, R_x, p_3, p_3' >$.

With respect to each application, we define a *context pool* as the collection of all contexts interesting to this application. For application App_room, which considers only rooms x and y, its context pool is P = $\{R_x, R_y\}$. Many existing context-aware middleware infrastructures or frameworks [15]–[17] support such pool-alike data structures for applications. Maintaining such a context pool helps applications to access their interesting contexts when necessary and apply collected context changes to corresponding contexts.

Contexts in a context pool can be inaccurate, incomplete or even conflicting with each other due to sensing noise or reasoning imprecision [4]. Since there is no precise oracle to generally validate contexts' correctness directly [10], one can check contexts against consistency constraints as mentioned earlier, so as to detect context inconsistencies for applications. *Consistency constraints* are typically expressed using a first-order logic (FOL) based constraint language [6] as follows:

$f := \forall v \in C\ (f) \mid \exists v \in C\ (f) \mid$
(f) and $(f) \mid (f)$ or $(f) \mid (f)$ implies $(f) \mid$ not $(f) \mid$
$bfunc(v_1, v_2, ..., v_n) \mid$ True $\mid$ False.

C represents a specific context in an application's context pool and v_i is a variable, which takes any element in a context as its value. Terminal $bfunc$ is a user-defined function that takes elements from a context as input and returns a boolean value (True or False). It serves as a predicate on values of contexts. For application App_room, its consistency constraint S_{loc} "no one can stay in rooms x and y at the same time" can be expressed as:

S_{loc}: $\forall v_1 \in R_x$ (not $(\exists v_2 \in R_y\ (equal(v_1,\ v_2))))$.

Consistency constraints are usually derived from general physical laws or application-specific requirements [1], [6], [8].

B. Context Constraint Checking Techniques

Context constraint checking techniques take contexts in a context pool and given consistency constraints as input, and return the result on whether any constraint has been violated and how the violation has occurred (i.e., *whether* and *how* questions). If there is no violation, the whether and how questions obtain a (False, {}) result. Otherwise, the question obtains a (True, $\{\text{inc}_1, \text{inc}_2, \ldots\}$) result. Each inconsistency inc_i is represented by a *link* [1], [6], connecting specific elements and their corresponding variables, explaining why a concerned constraint has been violated. Take an existing constraint checking technique PCC [6] as an example, and apply it to the scenario in Fig. 1. For the aforementioned constraint S_{loc}, PCC would report inc_1 upon context change chg_3. Inconsistency inc_1 is represented by a link $\{(v_1, p_1), (v_2, p_1)\}$, indicating that "$p_1$ in R_x" and "p_1 in R_y" together cause the violation to consistency constraint S_{loc}. Similarly, upon context change chg_7, PCC would report $\{(v_1, p_2), (v_2, p_2)\}$ and $\{(v_1, p_3), (v_2, p_3)\}$, i.e., inc_2 and inc_3, as shown in Fig. 1.

To facilitate subsequent discussions, we define *constraint checking result* (CCR) and *inconsistency detection result* (IDR). As mentioned earlier, CCR represents inconsistencies for the current contexts with all changes so far applied, and IDR represents all inconsistencies ever detected so far. Given a consistency constraint s and a context pool P, we use CCR(s, P) to denote the constraint checking result when checking contexts in P against constraint s. Take the scenario in Fig. 1 as an example. Let the context pool after applying change chg_i be P_i. Then CCR(S_{loc}, P_3) = $\{\text{inc}_1\}$ and CCR(S_{loc}, P_7) = $\{\text{inc}_2, \text{inc}_3\}$. We further explain IDR later.

In this paper, we consider four representative context constraint checking techniques, namely, ECC [1], PCC [6], Con-C [7] and GAIN [5]. ECC is the baseline, which checks contexts against consistency constraints non-incrementally. PCC works incrementally by reusing previous checking results. Con-C and GAIN check contexts in parallel by multiple CPU threads and GPU threads, respectively. Although the four techniques have different levels of efficiency, they follow the same constraint checking process and return the same checking result.

C. Problem Formulation

To detect context inconsistencies for applications, constraint checking techniques are scheduled repeatedly according to a specific scheduling strategy, e.g., aforementioned ImmedSched and BatchSched. Here, we use IDR to represent all context inconsistencies ever detected so far (even if they may be gone or resolved later).

Suppose that the whole context inconsistency detection starts at time point t_0 and each context change is collected at a distinct time point (i.e., chg_i is collected at time point t_i). Let the updated context pool at time point t_i after applying chg_i be P_i and the consistency constraint under consideration be s.

ImmedSched applies constraint checking upon each context change (i.e., checking happens at time points t_1, t_2, ...), and its IDR at time point t_m is the union of all previous CCR values until t_m (i.e., IDR(m) = $\cup_{i=1}^{m}$CCR(s, P_i)). Take the scenario in Fig. 1 as an example. Its final IDR by ImmedSched is $\{\text{inc}_1, \text{inc}_2, \text{inc}_3\}$ at time point t_9 (i.e., IDR(9) = $\cup_{i=1}^{9}$CCR(s, P_i)).

However, if one applies BatchSched to schedule constraint checking upon every k context changes (i.e, batch window size = k), its IDR at time point t_m is IDR(m) = $\cup_{i=1}^{m/k}$CCR(s, P_{i*k})) ($m \geq 2$). For example, if one sets the window size to be two (i.e., k = 2) as shown in Fig. 1, BatchSched's IDR at time point t_9 is IDR(9) = IDR(8) = $\cup_{i=1}^{4}$CCR(s, P_{i*2})) = $\{\text{inc}_2\}$.

ImmedSched's IDR is always *complete* since it schedules constraint checking upon every context change, thus capturing each potential inconsistency. However, BatchSched's IDR may miss context inconsistencies as illustrated by the preceding example. Suppose that there are two context changes chg_a

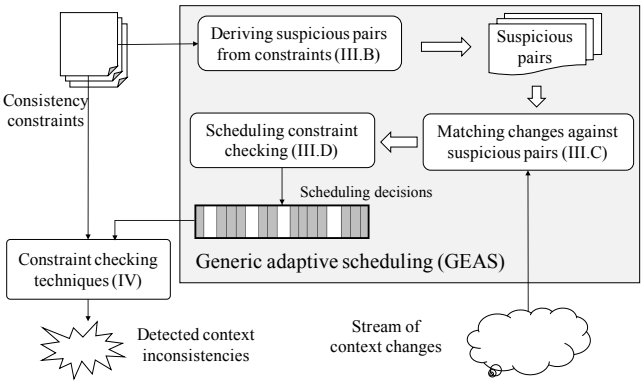

Fig. 2: Overview of our GEAS approach.

and chg_b ($a < b$) in one batch and the CCR value before checking this batch of changes is CCR(s, P_0). After studying relations between CCR and IDR values, we observe that if the corresponding CCR values upon changes chg_a and chg_b have: CCR(s, P_a) − CCR(s, P_0) $\neq \emptyset$ and CCR(s, P_a) − CCR(s, P_b) $\neq \emptyset$, then BatchSched's IDR upon this batch of changes could differ from that of ImmedSched. This is because checking upon change chg_a would detect new inconsistencies as compared to the situation before checking any change in this batch, and checking upon changes chg_a and chg_b together could cause such inconsistencies undetectable. Note that such undetectable inconsistencies due to checking in a batch (i.e., chg_a and chg_b together) may not necessarily be those new inconsistencies due to checking chg_a individually (i.e., without chg_b together), but such possibility does exist. For example, BatchSched in Fig. 1 checks upon change chg_4 (i.e., chg_3 and chg_4 in one batch) and thus misses inconsistency inc_1, which can otherwise be detected by ImmedSched to check upon chg_3 and chg_4 in turn, i.e., $\text{IDR}_{BatchSched}(4) \neq \text{IDR}_{ImmedSched}(4)$.

In this paper, we propose our GEAT approach to avoiding leaving the preceding problematic change pairs (named *suspicious pairs*) in one batch, so as to prevent context inconsistencies from being missed in the detection, i.e., $\text{IDR}_{GEAS}(i) = \text{IDR}_{ImmedSched}(i)$ always holds.

III. Generic Adaptive Scheduling

In this section, we elaborate on our GEAS approach, which is able to adaptively tune the batch window to avoid missing any inconsistency in IDR.

A. Approach Overview

Our GEAS approach aims to identify such context change pairs, which can possibly cause context inconsistencies missed in the detection when they are checked in one batch, and to isolate them into different batches by actively tuning the batch window size.

Consider our illustrated example in Fig. 1. GEAS would identify in advance that some context change pairs (e.g., "?? enters room x" and "?? leaves room x", or "?? enters room y" and "?? leaves room y") are suspicious and thus should not be scheduled in one batch. It then schedules constraint checking upon context changes chg_3 and chg_7 only, and obtains $\text{IDR}(3) = \{\text{inc}_1\}$ and $\text{IDR}(7) = \{\text{inc}_1, \text{inc}_2, \text{inc}_3\}$. By doing so, GEAS tunes its batch window size adaptively (size = 3 and 4 in turn) and its final IDR equals to $\text{IDR}_{ImmedSched}9 = \{\text{inc}_1, \text{inc}_2, \text{inc}_3\}$. Besides, GEAS schedules constraint checking only twice, thus improving the efficiency significantly. Compared with BatchSched, GEAS conducts constraint checking even less but obtains always complete IDR values, while BatchSched (window size = 2) results in its final IDR value of 66.7% missed rate.

We illustrate our GEAS approach in Fig. 2. GEAS consists of three phases. The first phase deviates suspicious pairs from given consistency constraints statically. The second phase matches collected context changes against our deviated suspicious pairs dynamically to judge whether scheduling a specific context change in the current batch could possibly cause context inconsistencies missed in the detection (i.e., making $\text{IDR}_{GEAS}(i) = \text{IDR}_{ImmedSched}(i)$ no longer hold). Finally, the third phase schedules context changes smartly by tuning the batch window size according to our matching results. In the following, we introduce the three phases in turn (III.B, III.C and III.D).

B. Deriving Suspicious Pairs from Consistency Constraints

GEAS' first phase derives suspicious pairs from given consistency constraints. To do so, we first introduce the impact that can be caused by a context change.

If a context change can possibly cause a consistency constraint to change its true value from True to False, such changes can potentially cause new context inconsistencies. Then, we say that such context changes have the *inc+ impact*. On the contrary, if a context change can possibly cause a consistency constraint to change its true value from False to True, such changes can potentially cause existing context inconsistencies undetectable if checked with earlier changes. Similarly, we say that such context changes have the *inc− impact*. In subsequent discussions, we use a short form of $<type, context>$ to represent a context change, as other fields are unnecessary for our analysis. We define some concepts below.

Definition 1 (**inc+ change**): If a context change has the inc+ impact only, it is classified as an *inc+ change*, indicating that this context change can potentially cause new inconsistencies.

Definition 2 (**inc− change**): If a context change has the inc− impact only, it is classified as an *inc− change*, indicating that this context change can potentially cause existing inconsistencies undetectable.

Definition 3 (**inc? change**): If a context change has both the inc+ impact and inc− impact, it is classified as an *inc? change*, indicating that this context change can potentially cause new inconsistencies, existing inconsistencies undetectable, or both.

We then discuss how to derive inc+/inc−/inc? changes for given consistency constraints. Consider consistency constraint s given by a universal formula $\forall v \in C\ (f)$ and three context changes $<+, C>$, $<-, C>$ and $<\#, C>$. For constraint s,

TABLE I: Deduction rules for classifying inc+, inc− and inc? changes.

Formula type	Deduction rules		
	Set of inc+ changes	**Set of inc− changes**	**Set of inc? changes**
$\forall v \in C\ (f)$	$Set_{inc+}(f)\ \cup\ \{<+, C>\}$	$Set_{inc-}(f) \cup \{<-, C>\}$	$Set_{inc?}(f) \cup \{<\#, C>\}$
$\exists v \in C\ (f)$	$Set_{inc+}(f)\ \cup\ \{<-, C>\}$	$Set_{inc-}(f) \cup \{<+, C>\}$	$Set_{inc?}(f) \cup \{<\#, C>\}$
(f_1) and (f_2)	$Set_{inc+}(f_1) \cup Set_{inc+}(f_2)$	$Set_{inc-}(f_1) \cup Set_{inc-}(f_2)$	$Set_{inc?}(f_1) \cup Set_{inc?}(f_2)$
(f_1) or (f_2)	$Set_{inc+}(f_1) \cup Set_{inc+}(f_2)$	$Set_{inc-}(f_1) \cup Set_{inc-}(f_2)$	$Set_{inc?}(f_1) \cup Set_{inc?}(f_2)$
(f_1) implies (f_2)	$Set_{inc-}(f_1) \cup Set_{inc+}(f_2)$	$Set_{inc+}(f_1) \cup Set_{inc-}(f_2)$	$Set_{inc?}(f_1) \cup Set_{inc?}(f_2)$
not (f)	$Set_{inc-}(f)$	$Set_{inc+}(f)$	$Set_{inc?}(f)$
$bfunc(v_1, ...)$	$\emptyset$	$\emptyset$	$\emptyset$

$<+, C>$ is an inc+ change. This is because adding a new element into C can possibly cause the universal formula from satisfied to violated (i.e., truth value changes from True to False), while making the truth value change from False to True is impossible. Similarly, $<-, C>$ is an inc− change, because deleting an existing element from C can possibly cause the universal formula from violated to satisfied (i.e., truth value changes from False to True), while making the truth value change from True to False is impossible. $<\#, C>$ is an inc? change, because $<\#, C>$ can change any element's value arbitrarily and thus cause an unpredictable impact.

Following this principle, we present our deduction rules in Table I, to derive three sets of classified context changes for a given consistency constraint s. The three sets are $Set_{inc+}(s)$, $Set_{inc-}(s)$ and $Set_{inc?}(s)$.

For example, consider the aforementioned S_{loc} constraint in Section II.A. $Set_{inc+}(S_{loc})$ can be derived as follows: $Set_{inc+}(S_{loc}) = Set_{inc+}(\text{not}\ (\exists v_2 \in R_y\ (equal(v_1,\ v_2)))) \cup \{<+, R_x>\} = Set_{inc-}(\exists v_2 \in R_y\ (equal(v_1,\ v_2))) \cup \{<+, R_x>\} = Set_{inc-}(equal(v_1,\ v_2)) \cup \{<+, R_y>\} \cup \{<+, R_x>\} = \emptyset \cup \{<+, R_y>\} \cup \{<+, R_x>\} = \{<+, R_x>, <+, R_y>\}$.

Similarly, its two other sets of inc− and inc? changes are as follows: $Set_{inc-}(S_{loc}) = \{< -, R_x >, < -, R_y >\}$ and $Set_{inc?}(S_{loc}) = \{< \#, R_x >, < \#, R_y >\}$.

Based on the three derived sets of inc+, inc− and inc? changes, we define *suspicious pairs*.

Definition 4 (Suspicious pair): A *suspicious pair* is a combination of two context changes. The first change has the inc+ impact and the second has the inc− impact.

According to the definition, a suspicious pair can be of four forms, namely, (inc+, inc−), (inc+, inc?), (inc?, inc−) and (inc?, inc?).

Then, all suspicious pairs for this constraint S_{loc} are as follows (superscript denotes specific type of change, for illustration only):

$(<+,\ R_x/R_y>^{inc+}, <-,\ R_x/R_y>^{inc-})$,
$(<+,\ R_x/R_y>^{inc+}, <\#,\ R_x/R_y>^{inc?})$,
$(<\#,\ R_x/R_y>^{inc?}, <-,\ R_x/R_y>^{inc-})$,
$(<\#,\ R_x/R_y>^{inc?}, <\#,\ R_x/R_y>^{inc?})$.

By doing so, suspicious pairs can be derived for a given consistency constraint s automatically, and each suspicious pair contains two context changes in order, e.g., chg_a and chg_b. The first changes chg_a can possibly cause new inconsistencies when calculating a CCR value, and the second one chg_b can possibly cause existing inconsistencies undetectable. Therefore, changes chg_a and chg_b can together cause CCR(s, P_a) − CCR(s, P_0) $\neq \emptyset$ and CCR(s, P_a) − CCR(s, P_b) $\neq \emptyset$. Here, we use P_0 to represent the context pool before applying changes chg_a and chg_b. As a result, the new inconsistencies in CCR(s, P_a) − CCR(s, P_0), detectable by scheduling constraint checking upon chg_a, can possibly become undetectable if one schedules changes chg_a and chg_b in one batch (i.e., not scheduling constraint checking upon chg_a).

C. Matching Context Changes Against Suspicious Pairs

GEAS' second phase matches collected context changes against derived suspicious pairs, so as to judge whether scheduling the next change in the current batch could possibly cause inconsistencies missed in the detection. In the matching, we examine each pair formed by an existing context change in the current batch and the next context change.

Take the context changes in Fig. 1 as an example. Upon change chg_4, GEAS examines three pairs ($<chg_1,\ chg_4>$, $<chg_2,\ chg_4>$, $<chg_3,\ chg_4>$) in turn, to judge whether any change pair matches any suspicious pair derived for the consistency constraint under consideration. Since chg_3 is an inc+ change and chg_4 is an inc− change, their pair $<chg_3, chg_4>$ matches one suspicious pair ($<+,\ R_y>, <-,\ R_x>$). In other words, context changes chg_3 and chg_4 together can cause inconsistencies undetectable if scheduled in one batch for constraint checking. In fact, we indeed observe that scheduling chg_3 and chg_4 in one batch causes inconsistency inc_1 missed in an IDR calculation as shown in Fig. 1. Its underlying reason is: CCR(s, P_3) − CCR(s, P_2) $\neq \emptyset$ and CCR(s, P_3) − CCR(s, P_4) $\neq \emptyset$, as we analyzed earlier.

We present the matching algorithm as Algorithm 1. Given any new context change, it is attached to each context change in the current batch in turn (Line 1) to examine whether their combination matches any suspicious pair derived in advance (Line 3). In order to improve the efficiency, the algorithm

Algorithm 1 Matching changes against suspicious pairs

Input: S (all suspicious pairs), *chg* (a context change), *setOfChgs* (context changes in the current batch)
Output: *result* (whether a suspicious pair is matched)

```
1: for any change c in setOfChgs do
2:     if c.category == "inc+" or "inc?" then
3:         if S contains (c, chg) then
4:             result := TRUE;
5:             break;
6:         end if
7:     end if
8: end for
9: result := FALSE;
10: return result
```

Algorithm 2 Scheduling constraint checking

Input: Q (queue maintained), S (all suspicious pairs), *chg* (a context change)

```
1: let setOfChgs contain all changes in the current batch Q;
2: if suspPairMatch(S, chg, setOfChgs) returns TRUE then
3:     Check(setOfChgs); // Schedule constraint checking
4:     Q.clear(); // Make the current batch empty
5: end if
6: decide change chg's category, e.g., inc+, inc− or inc?;
7: Q.add(chg);
8: return
```

attaches the new context change to inc+ and inc? changes only (Line 2), since the first context change in a suspicious pair must be an inc+ change or an inc? change according to the definition. To do so, each change maintains a *category* field to indicate its specific type of change, e.g., inc+, inc− or inc?.

D. Scheduling Constraint Checking

GEAS' final phase schedules constraint checking adaptively according to its matching results, so as to avoid missing any context inconsistency in the detection. We present the scheduling algorithm as Algorithm 2.

The algorithm maintains the current batch queue Q, and any newly collected context change may be added into this queue in order if necessary. Upon any new context change, a new pair formed by an existing change in Q and this new change is examined by Algorithm 1 (Line 2, by suspPairMatch), to find out whether scheduling this change in the current batch can possibly cause any inconsistency missed in the detection. If yes, the algorithm schedules constraint checking (Line 3) immediately on the current batch of context changes, and then clears them (Line 4) after that. Then, no matter constraint checking has been scheduled or not, the new context change obtains its category value (e.g., set as "inc+" if it is an inc+ change) and is added into Q (Line 7).

Such a matching and scheduling process continues until all context changes are processed. In this way, GEAS automatically tunes its batch window size according to matching results and schedules constraint checking only when necessary, thus both improving the efficiency and avoiding missing context inconsistencies in the detection.

IV. Existing Context Constraint Checking Techniques

GEAS is generic in the sense that it can easily work with existing context constraint checking techniques (e.g., ECC [1], PCC [6], Con-C [7] and GAIN [5]) with no or little adjustment.

We partition existing constraint checking techniques into two categories: non-cache-based and cache-based. A constraint checking technique is *non-cache-based* if it supports processing multiple context changes together (not relying on cached previous checking results), and *cache-based* if it does not support (e.g., can only process context changes one by one, due to the dependency on cached previous checking results). Existing context constraint checking techniques such as ECC, Con-C and GAIN belong to the non-cache-based category as they work non-incrementally, and PCC belongs to the cache-based category as it works incrementally.

For non-cache-based context constraint checking techniques, GEAS is directly applicable, since such techniques are already ready for processing multiple context changes in one batch scheduled by GEAS. In fact, we successfully applied GEAS to ECC, Con-C and GAIN without any adjustment to their checking semantics.

For a cache-based context constraint checking technique like PCC, one needs to make a little adjustment to its checking semantics, since such technique was originally designed for processing one context change a time (that is why it is incremental). For PCC, we modified its checking semantics a little bit to support processing multiple context changes in one batch and name its new version MPCC. We list the new truth value evaluation and link generation semantics for universal formula in Fig. 3, as an example.

Fully understanding such semantics needs some knowledge on truth value evaluation and link generation (for answering the whether and how question, respectively). Here, we very briefly highlight our ideas for the modification. PCC processes a context change as one of four cases, namely, fully reusable (the change does not affect C and sub-formula f at all), processing a single addition change (affecting C), processing a single deletion change (affecting C), and processing a change that affects sub-formula f, incurring different levels of reusability of previous checking results. Here, C represents the current C value and C_0 represents the last C value. Similarly, $\mathcal{T}/\mathcal{L}$ represents the new truth value/generated links and $\mathcal{T}_0/\mathcal{L}_0$ represents the last truth value/generated links. MPCC needs to process multiple context changes a time, and thus it partitions the processing into four cases as well. The first and last cases are the same as before, but the second case is for the situation where there are multiple addition changes only (i.e., no deletion change), and the third case is for the situation where there is at least one deletion change (i.e., multiple deletion changes only, or both addition and deletion changes). Such modifications are immaterial to the checking semantics and thus are straightforward to apply. We ensure MPCC's equivalence to PCC during the modifications.

$\mathcal{T}[\forall v \in C(f)]_\alpha =$

1) $\mathcal{T}_0[\forall v \in C(f)]_\alpha$,
   if C has no change (i.e., $C = C_0$) and $\mathsf{affected}(f) = \bot$;
2) $\mathcal{T}_0[\forall v \in C(f)]_\alpha \wedge \mathcal{T}[f]_{\mathsf{bind}((v,x_1),\alpha)} \wedge \ldots \wedge \mathcal{T}[f]_{\mathsf{bind}((v,x_n),\alpha)} \mid x_i \in C - C_0$,
   if C has addition changes only;
3) $\top \wedge \mathcal{T}_0[f]_{\mathsf{bind}((v,x_1),\alpha)} \wedge \ldots \wedge \mathcal{T}_0[f]_{\mathsf{bind}((v,x_m),\alpha)} \wedge \mathcal{T}[f]_{\mathsf{bind}((v,y_1),\alpha)} \wedge \ldots \wedge \mathcal{T}[f]_{\mathsf{bind}((v,y_n),\alpha)} \mid x_i \in C_0 \cap C,\ y_i \in C - C_0$,
   if C has any deletion change (deletion changes only, or both addition and deletion changes);
4) $\top \wedge \mathcal{T}[f]_{\mathsf{bind}((v,x_1),\alpha)} \wedge \ldots \wedge \mathcal{T}[f]_{\mathsf{bind}((v,x_n),\alpha)} \mid x_i \in C$,
   if $\mathsf{affected}(f) = \top$.

$\mathcal{L}[\forall v \in C(f)]_\alpha =$

1) $\mathcal{L}_0[\forall v \in C(f)]_\alpha$,
   if C has no change (i.e., $C = C_0$) and $\mathsf{affected}(f) = \bot$;
2) $\mathcal{L}_0[\forall v \in C(f)]_\alpha \cup \{l | l \in \{(\mathsf{violated}, \{(v, x_i)\})\} \otimes \mathcal{L}[f]_{\mathsf{bind}((v,x_i),\alpha)}\} \mid x_i \in C - C_0 \wedge \mathcal{T}[f]_{\mathsf{bind}((v,x_i),\alpha)} = \bot$,
   if C has addition changes only;
3) $\{l | l \in \{(\mathsf{violated}, \{(v, x_i)\})\} \otimes \mathcal{L}_0[f]_{\mathsf{bind}((v,x_i)\alpha)}\} \cup \{l | l \in \{(\mathsf{violated}, \{(v, y_i)\})\} \otimes \mathcal{L}[f]_{\mathsf{bind}((v,y_i)\alpha)}\} \mid x_i \in C_0 \cap C \wedge \mathcal{T}[f]_{\mathsf{bind}((v,x_i),\alpha)} = \bot,\ y_i \in C - C_0 \wedge \mathcal{T}[f]_{\mathsf{bind}((v,y_i),\alpha)} = \bot$,
   if C has any deletion change (deletion changes only, or both addition changes and deletion changes);
4) $\{l | l \in \{(\mathsf{violated}, \{(v, x_i)\})\} \otimes \mathcal{L}[f]_{\mathsf{bind}((v,x_i),\alpha)}\} \mid x_i \in C \wedge \mathcal{T}[f]_{\mathsf{bind}((v,x_i),\alpha)} = \bot$,
   if $\mathsf{affected}(f) = \top$.

Fig. 3: Truth value evaluation and link generation for universal formula in MPCC.

V. Evaluation

In this section, we evaluate GEAS' efficiency and effectiveness and compare it with existing work. We also evaluate GEAS' practical performance in real-world scenarios.

A. Research Questions (RQ1-3)

We aim to answer the following three research questions.

RQ1: *How serious is the inconsistency-missing problem with BatchSched for checking context consistency?*

RQ2: *How efficient is GEAS in context inconsistency detection, as compared with existing work (i.e., ImmedSched and BatchSched)?*

RQ3: *How effective is GEAS in terms of avoiding missing context inconsistencies in the detection, as compared with existing work (i.e., BatchSched)?*

Answering RQ1 motivates our GEAS work. Answering RQ2 validates how GEAS improves the efficiency for context inconsistency detection. Answering RQ3 validates how GEAS avoids missing context inconsistencies in the detection. They together justify GEAS' necessity and usefulness.

B. Experimental Design and Setup

To answer the three research questions, we design two dependent variables. *Checking time* is for measuring the efficiency of context inconsistency detection (processing all context changes). *Inconsistency missing rate* (or *missing rate* for short) is for measuring the effectiveness of context inconsistency detection, which is defined by the proportion of detected context inconsistencies against all inconsistencies in theory.

With respect to these dependent variables, we design the following three independent variables, whose settings can affect the measurement of our dependent variables:

- *Scheduling strategy.* We considered three scheduling strategies: ImmeSched, BatchSched and our GEAS.
- *Batch window size.* We controlled different sizes for the batch window when applying BatchSched.
- *Constraint checking technique.* We considered four state-of-the-art context constraint checking techniques, namely, ECC [1], PCC [6] (by MPCC), Con-C [7] and GAIN [5]. They can all work with the three scheduling strategies.

For research question RQ1, we measure the checking time and missing rate for BatchSched in context inconsistency detection. We controlled the batch window size from one to six. The upper bound was decided by the fact that BatchSched caused a quickly increasing missing rate when the window size grew, and that the rate was already above 60% at size = 6, which is undesirable and suggests that BatchSched can hardly be useful in practice with an even larger window size. For research question RQ2, we measure the checking time under 12 strategy-technique combinations (three scheduling strategies and four constraint checking techniques). For research question RQ3, we measure the missing rate also under the aforementioned 12 strategy-technique combinations.

We selected as our experimental subject the SUTPC application, which was from existing work on evaluating context constraint checking techniques ECC, PCC, Con-C and GAIN. The application is accompanied with 21 consistency constraints and 1.6 million 24-hour taxi data, concerning the checking of taxis' GPS location, driving speed, direction, service status, and so on. These data concern 760 taxis from one company in a city in China. Context changes were derived from the taxi data and their quantity (over 4 million) relied on the application's functions, e.g., planning an optimal driving route or monitoring a hot area's traffic conditions. In experiments,

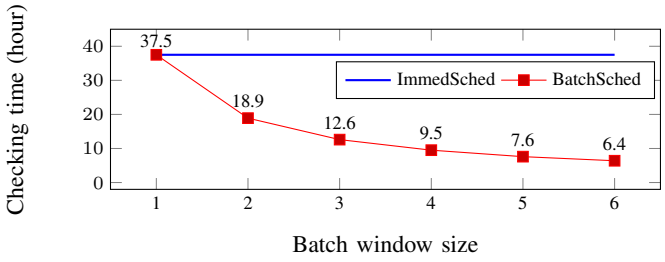

Fig. 4: Checking time comparison for BatchSched.

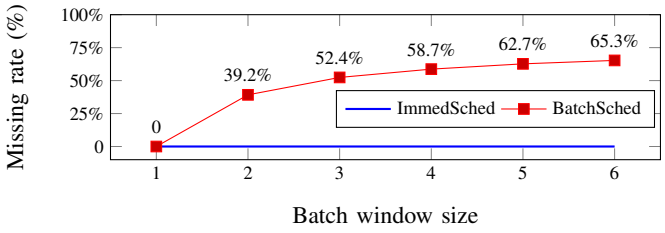

Fig. 5: Missing rate comparison for BatchSched.

we controlled to process these context changes in sequence and proceed only when previous checking completed. This may cause a low-efficiency technique (e.g., ECC) to spend over 24 hours to process all changes, but fully respects the technique's nature as no change would be skipped in the checking (i.e., no buffer overflow due to too slow processing). As a result, different constraint checking techniques would only cause different efficiency levels, but would not affect the measurement of the missing rate, which is fully decided by the scheduling strategy (our focus).

All experiments were conducted on a desktop PC with an Intel® Core^{TM} i7-6700 CPU @3.40 GHz and 16GB RAM. This machine is installed with MS Windows 10 Professional and Oracle Java 8.

C. Experimental Results and Analyses

In the following, we answer the preceding three research questions in turn.

RQ1 (**motivation**). We conducted experiments with different batch window sizes for BatchSched. As the efficiency of a specific constraint checking technique does not affect the missing rate in our experiments as mentioned earlier, we fixed the technique to be ECC. Using ImmedSched's data as the baseline, we report BatchSched's performance data (checking time and missing rate) in Fig. 4 and Fig. 5.

The blue lines represent ImmedSched's checking time and missing rate, whose values are not affected by the batch window size. The red curves represent BatchSched's checking time and missing rate, whose values exhibit a steadily decreasing and increasing trend, respectively. Since ImmedSched checked contexts upon each change, it detected all context inconsistencies, accounting for a missing rate of 0%. However, when it comes to BatchSched, we observe that with the growth of the window size, the checking time was indeed reduced (up to 83.0%), but the missing rate instead increased rapidly (up to 65.3%). BatchSched worked faster than ImmedSched since it was able to merge the processing of multiple context changes (one to six) and this saved a lot of redundant checking overhead. However, it also caused a high missing rate since some context inconsistencies became undetectable if certain context changes were processed in one batch.

We conclude our answer to research question RQ1:

> *BatchSched could improve the efficiency of context inconsistency detection, but also caused a serious inconsistency-missing problem. Its missing rate could be up to 65.3% when its batch window size was 6.*

RQ2 (**efficiency**). We measured the checking time under 12 strategy-technique combinations (three scheduling strategies, namely, ImmedSched, BatchSched and GEAS, and four constraint checking techniques, namely, ECC, PCC, Con-C and GAIN). We use BatchSched-x to represent applying BatchSched with a batch window size of x. By answering RQ1, we observe that the missing rate could be up to 65.3% when the window size was six, which suggests that BatchSched can hardly be useful in practice with an even larger window size. Therefore, we in answering RQ2 consider three window sizes, i.e., 2, 4, 6, to be representative (1 is omitted as it makes the scheduling reduced to ImmedSched). We also consider BatchSched with a batch window size of the value averaged from all window sizes adaptively tuned by GEAS. The value was 4.9 and thus we made it 5 for integer. The corresponding schedule is named BatchSched-mimic. We report efficiency data (checking time) in Fig. 6 and Fig. 7.

The checking time is compared in a normalized way (ECC data as 100%) and in hour, respectively, in Fig. 6 and Fig. 7. We observe in Fig. 7 that GEAS reduced the checking time significantly for constraint checking techniques, in particular for ECC (up to 84.5%). This is because ECC is most inefficient by checking in a non-incremental and non-parallel way, and GEAS helped it most significantly by compressing the checking of extremely a lot of context changes in a batch way. The efficiency improvement brought by GEAS for PCC is not obvious in Fig. 7, since PCC itself is already the most efficiency checking technique among all.

We observe in Fig. 6 that GEAS reduced the checking time greatly for all constraint checking techniques, e.g., 84.5% reduction for ECC, 84.4% for Con-C, 83.5% for GAIN and 29.9% for PCC. In other words, compared with ImmedSched, all four constraint checking techniques were improved in efficiency when equipped with our GEAS, i.e., 645% improvement for ECC, 641% for Con-C, 606% for GAIN and 143% for PCC. What is worth noticing is that although BatchSched-mimic used the mimicked batch window size (5), slightly larger than GEAS' averaged window size (4.9), its efficiency was still inferior than that of GEAS.

We conclude our answer to research question RQ2:

> *GEAS was efficient in context inconsistency detection. It achieves 143–645% efficiency improvement compared with ImmedSched, when equipped with existing constraint checking techniques, e.g., ECC, PCC, Con-C and GAIN.*

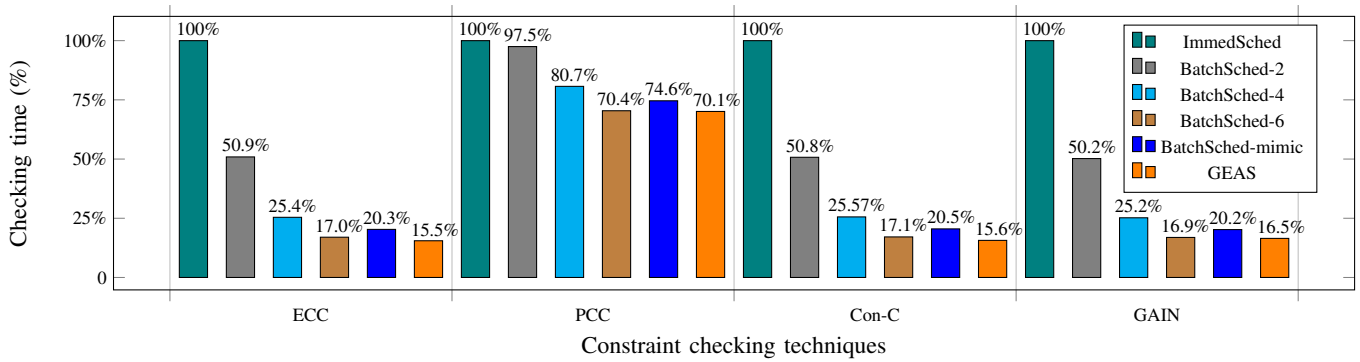

Fig. 6: Efficiency (checking time) comparison (normalized with ECC data as 100%).

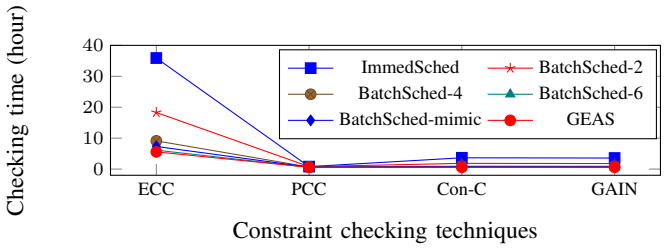

Fig. 7: Efficiency (checking time) comparison (in hour).

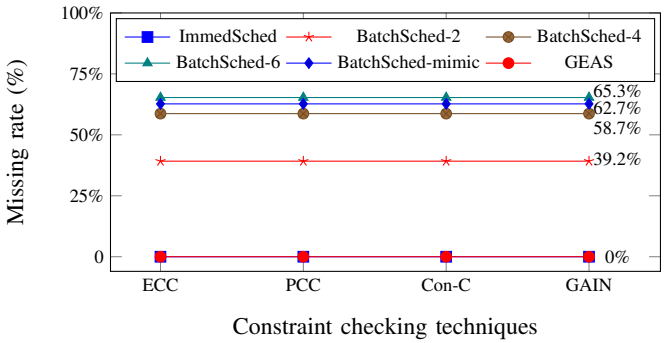

Fig. 8: Missing rate comparison.

RQ3 (**effectiveness**). We finally evaluate the missing rate for context inconsistency detection. Since we controlled the processing of context changes in sequence to proceed only when previous checking completed, we would be able to obtain the theoretical upper bound (e.g., all context inconsistencies), which is used for calculating the missing rate. Also, due to such experimental controlling, the missing rate is decided only by the scheduling strategy, no matter which constraint checking technique is used, as shown in Fig. 8. We observe that GEAS missed zero context inconsistency, as ImmedSched did. Compared to GEAS, BatchSched caused enormous context inconsistencies missed (39.2–65.3%), and such inconsistency-missing problem would seriously impair the correct functioning of applications running on the contexts.

What is worth mentioning is that although BatchSched-mimic adopted a batch window of size close to what was averaged for GEAS, it only achieved similar checking efficiency (still a little bit inferior), while its effectiveness was clearly worsen than GEAS by largely missing context inconsistencies (missing rate of 62.7%). This is because it is not the batch window size that avoids missing inconsistencies in the detection, but how the window is tuned does, as we analyzed for the GEAS design earlier.

We conclude our answer to research question RQ3:

GEAS was effective in context inconsistency detection. It completely avoided missing context inconsistencies in the detection, while BatchSched caused 39.2–65.3% missing rate.

D. Real-world Scenarios

We briefly report GEAS' performance in real-world scenarios. We used the same taxi data, but had all strategy-technique combinations process context changes according to actual timestamps associated with the data. This setting would no longer guarantee context changes not skipped (if processed too slowly, e.g., for ECC). We observe that GEAS still guaranteed its zero missing rate, combined with any of the four constraint checking techniques, while ImmedSched caused 0–95.4% missing rate and it is 39.2-67.7% for BatchSched.

Besides, we controlled to increase the workload by duplicating context changes multiple times. Under the same setting for rush-hour taxi data, ImmedSched caused 93.1–97.8% missing rate; BatchSched achieved 100–689% efficiency improvement as compared to ImmedSched, but caused 49.9–93.7% missing rate; our GEAS achieved 106–935% efficiency improvement and incurred zero missing rate for all constraint checking techniques except ECC. We note that in such a dynamic testing scenario, achieving a zero missing rate is probably impossible, as network connection and data parsing also took time and this caused checking not in right time, leading to some missed context inconsistencies.

In summary, we validate our GEAS' high efficiency and effectiveness both in controlled experiments and under practical settings.

E. Threats to Validity

Our experiments chose only one application and this can cause the threat to external validity of our conclusion. We alleviate this threat by three efforts. First, the application, as well as its accompanied taxi data and consistency constraints,

were from existing work on context constraint checking. Second, the data set itself is huge, containing 1.6 million raw data and over 4 million corresponding context changes, alleviating the possibility of special data leading to experimental bias. Third, the taxi data have intervals varying in 20–3000 ms for consecutive data. Our experiments tried two ways to use the data, one by a static setting that controlled the checking to proceed only when previous checking completed, respecting each checking technique's characteristics (e.g., avoiding skipping context changes due to low efficiency), another by a dynamic setting that compared each alternative's potential along with real-world timestamps.

The internal validity of our conclusion may be threatened as our derived suspicious pairs are conservative. Thus, the efficiency results may not precisely reflect GEAS' true capability. Nevertheless, even with such conservation, GEAS still achieved amazing efficiency improvement. Moreover, we have formalized and proved GEAS' correctness.

VI. Related Work

Context-aware adaptive applications, e.g., ConChat [18], ActiveCampus [19], SeNIE [20], TourApp [21], PhoneAdapter [22], Locale [23] and Navia [24], are gaining increasing research attention. Various application frameworks and middleware infrastructures, e.g., CARISMA [15], Gaia [25], EgoSpaces [16], Lime [26], Cabot [6], CAPPUCINO [27], Disnix [28] and Adam [29] have been proposed to support context-aware programming and management of context data. Increasing research work has identified additional challenges in developing such applications safely and correctly [29]–[31].

One line of research work focusing on quality guarantee for context-aware adaptive applications works along the context validation way. Some pieces of work directly focus on context inconsistency detection and resolution, aiming at identifying problematic contexts or context inconsistencies among large volumes of data and preventing applications from using them directly. Among them, the efficiency is a major concern. For example, ECC [1] checked contexts exhaustively, while PCC [6] did in an incremental way by reusing previous checking results. It achieves this by trading time with extra space. Con-C [7] and GAIN [5] checked contexts in parallel by exploiting multiple CPU and GPU threads, respectively. Their focus is how to improve the checking efficiency, while making the task scheduling always balanced. Detected context inconsistencies can then be resolved in a heuristic way [4], [32] or analytical way [3], [13], [33], such that applications can run over consistent contexts, not behaving abnormally.

There are also some pieces of work focusing on detecting inconsistencies in other software artifacts, e.g., XML documents [1], [8], [34], UML models [2], [35], data structures [3], workflows [13] and distributed source code [36]. Their concerned software artifacts typically change slowly or rarely, and thus do not have an emergent requirement on efficiency.

This work echoes some findings in existing work on editing script consistency [37] and unstable inconsistency suppression [10]. They all observe that earlier detected inconsistencies can become unstable or disappearing if detected at other time points. While the existing work focused on how to avoid such consequences, this work exploits this fact to smartly decide when to schedule constraint checking such that the efficiency can be improved. Besides, this work is also generic and can be easily applicable to existing constraint checking techniques, complementing and orthogonal to current efficient constraint checking efforts.

VII. Conclusion

In this paper, we studied the context inconsistency detection problem. Existing context constraint checking techniques tried different incremental and parallel ways to improve the checking efficiency, but they have to cooperate with the immediate scheduling strategy in order not to miss context inconsistencies. This ensures the complete detection, but also limits the efficiency. We addressed this problem by proposing our GEAS approach, which adaptively tunes its batch window size, such that the checking efficiency can be improved significantly, and at the same time ensure zero missing rate for context inconsistency detection.

Our GEAS approach is fully automated, achieving 143–645% efficiency improvement. It is also generic, applicable to existing context constraint checking techniques. Besides, we have also formally analyzed GEAS and proved its correctness.

Our GEAS currently works in a conservative way. It schedules constraint checking whenever adding a context change into the current batch can potentially cause any context inconsistency missed. We plan to further refine GEAS to make it more precise in this judgment. Also, we assume that all context changes are at distinct time points. We are now proving that processing context changes collected at the same time point does not affect GEAS' correctness as well, in avoiding missing context inconsistencies.

Acknowledgment

This work was supported in part by National Basic Research 973 Program (Grant No. 2015CB352202), and National Natural Science Foundation (Grant Nos. 61472174, 61690204) of China. The authors would also like to thank the support from the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

References

[1] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelsteiin, "xlinkit: A consistency checking and smart link generation service," *ACM Transaction on Internet Technology (TOIT)*, vol. 2, no. 2, pp. 151–185, may 2002.

[2] A. Egyed, "Instant consistency checking for the uml," in *Proceedings of 29th International Conference on Software Engineering (ICSE)*, Shanghai, China, May 2006, pp. 381–390.

[3] B. Demsky and M. C. Rinard, "Goal-directed reasoning for specification-based data structure repair," *IEEE Transactions on Software Engineering (TSE)*, vol. 32, no. 12, pp. 931–951, 2006.

[4] C. Xu, S. C. Cheung, W. K. Chan, and C. Ye, "Heuristics-based strategies for resolving context inconsistencies in pervasive computing applications," in *Proceedings of the 28th International Conference on Distributed Computing Systems (ICDCS)*, Beijing, China, Jun 2008, pp. 713–721.

[5] J. Sui, C. Xu, S. C. Cheung, W. Xi, Y. Jiang, C. Cao, X. Ma, and J. Lu, "Hybrid CPU–GPU constraint checking: Towards efficient context consistency," *Information and Software Technology (IST)*, vol. 74, pp. 230–242, 2016.

[6] C. Xu, S. C. Cheung, W. K. Chan, and C. Ye, "Partial constraint checking for context consistency in pervasive computing," *ACM Transaction on Software Engineering and Methodology (TOSEM)*, vol. 19, no. 3, pp. 9:1–9:61, Jan 2010.

[7] C. Xu, Y. Liu, S. C. Cheung, C. Cao, and J. Lu, "Towards context consistency by concurrent checking for internetware applications," *Science China Information Sciences*, vol. 56, no. 8, pp. 1–20, Aug 2013.

[8] S. P. Reiss, "Incremental maintenance of software artifacts," vol. 32, no. 9, pp. 113–122, 2006.

[9] C. Xu and S. C. Cheung, "Inconsistency detection and resolution for context-aware middleware support," in *Proceedings of the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE/ESEC)*, vol. 30, no. 5. Lisbon,Portugal: ACM, 2005, pp. 336–345.

[10] C. Xu, W. Xi, S. C. Cheung, X. Ma, C. Cao, and J. Lu, "Cina: Suppressing the detection of unstable context inconsistency," *IEEE Transactions on Software Engineering (TSE)*, vol. 41, no. 9, pp. 842–865, 2015.

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

[12] C. Xu, S. C. Cheung, W. K. Chan, and C. Ye, "On impact-oriented automatic resolution of pervasive context inconsistency," in *Proceedings of the Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE/ESEC)*, Dubrovnik, Croatia, Sep 2007, pp. 569–572.

[13] C. Chen, C. Ye, and H. A. Jacobsen, "Hybrid context inconsistency resolution for context-aware services," in *Proceedings of the 2011 IEEE International Conference on Pervasive Computing and Communications (PERCOM)*, Seattle, Washington, USA, Mar 2011, pp. 10–19.

[14] H. Lu, W. K. Chan, and T. H. Tse, "Testing pervasive software in the presence of context inconsistency resolution services," in *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, Leipzig, Germany, May 2008, pp. 61–70.

[15] L. Capra, W. Emmerich, and C. Mascolo, "Carisma: Context-aware reflective middleware system for mobile applications," *IEEE Transactions on Software Engineering (TSE)*, vol. 29, no. 10, pp. 929–945, 2003.

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

[17] M. Sama, D. S. Rosenblum, Z. Wang, and S. Elbaum, "Multi-layer faults in the architectures of mobile, context-aware adaptive applications," *The Journal of Systems and Software (JSS)*, vol. 83, no. 6, pp. 906–914, 2010.

[18] A. Ranganathan, R. H. Campbell, A. Ravi, and A. Mahajan, "Conchat: a context-aware chat program," *IEEE Pervasive Computing*, vol. 1, no. 3, pp. 51–57, 2002.

[19] W. G. Griswold, R. Boyer, S. W. Brown, and M. T. Tan, "A component architecture for an extensible, highly integrated context-aware computing infrastructure," in *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, Oregon, USA, 2003, pp. 363–372.

[20] M. Sama, V. Pacella, E. Farella, L. Benini, and B. Riccó, "3dID: A low-power, low-cost hand motion capture device," in *Proceedings of the Conference on Design, Automation and Test in Europe: Designers' Forum (DATE)*, ser. DATE'06. European Design and Automation Association, 2006, pp. 136–141.

[21] Z. Wang, S. Elbaum, and D. S. Rosenblum, "Automated generation of context-aware tests," in *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, ser. ICSE'07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 406–415.

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

[23] "Locale," http://www.twofortyfouram.com/.

[24] "Navia," http://induct-technology.com/.

[25] A. Ranganathan and R. H. Campbell, "An infrastructure for context-awareness based on first order logic," *Personal and Ubiquitous Computing (PUC)*, vol. 7, no. 6, pp. 353–364, 2003.

[26] A. L. Murphy, G. P. Picco, and G. C. Roman, "LIME: A coordination model and middleware supporting mobility of hosts and agents," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 15, no. 3, pp. 279–328, 2006.

[27] D. Romero, R. Rouvoy, L. Seinturier, S. Chabridon, D. Conan, and N. Pessemier, "Enabling context-aware web services: A middleware approach for ubiquitous environments," in *Enabling Context-Aware Web Services: Methods, Architectures, and Technologies*, M. Sheng, J. Yu, and S. Dustdar, Eds. Chapman and Hall/CRC, May 2010, pp. 113–135.

[28] S. van der Burg and E. Dolstra, "A self-adaptive deployment framework for service-oriented systems," in *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. New York, NY, USA: ACM, 2011, pp. 208–217.

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

[30] C. Q. Adamsen, G. Mezzetti, and A. Moller, "Systematic execution of android test suites in adverse conditions," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*, ser. ISSTA 2015. Baltimore, MD, USA: ACM, 2015, pp. 83–93.

[31] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek, "SIG-Droid: Automated system input generation for android applications," vol. 00, pp. 461–471, 2015.

[32] J. Lobo, J. Chomicki, and S. Naqvi, "Conflict resolution using logic programming," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 15, pp. 244–249, Jan/Feb 2003.

[33] Y. Xiong, Z. Hu, H. Zhao, H. Song, M. Takeichi, and H. Mei, "Supporting automatic model inconsistency fixing," in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (FSE/ESEC)*. ACM, 2009, pp. 315–324.

[34] C. Nentwich, W. Emmerich, A. Finkelsteiin, and E. Ellmer, "Flexible consistency checking," *ACM Transaction on Software Engineering and Methodology (TOSEM)*, vol. 12, no. 1, pp. 28–63, Jan 2003.

[35] X. Blanc, I. Mounier, A. Mougenot, and T. Mens, "Detecting model inconsistency through operation-based model construction," in *Proceedings of the 30th International Conference on Software Engineering (ICSE)*. Leipzig, Germany: ACM, May 2008, pp. 511–519.

[36] A. Demuth, M. Riedl-Ehrenleitner, and A. Egyed, "Efficient detection of inconsistencies in a multi-developer engineering environment," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Singapore, 2016, pp. 590–601.

[37] T. Kehrer, U. Kelter, and G. Taentzer, "Consistency-preserving edit scripts in model versioning," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Silicon Valley, California, USA, Nov 2013, pp. 191–201.