Generic Adaptive Scheduling for Efficient Context Inconsistency Detection

Huiyan Wang, *Student Member, IEEE,* Chang Xu, *Senior Member, IEEE,* Bingying Guo, Xiaoxing Ma, *Member, IEEE,* and Jian Lu

Abstract—Many applications use contexts to understand their environments and make adaptation. However, contexts are often inaccurate or even conflicting with each other (a.k.a. *context inconsistency*). To prevent applications from behaving abnormally or even failing, one promising approach is to deploy constraint checking to detect context inconsistencies. A variety of constraint checking techniques have been proposed, based on different incremental or parallel mechanisms for the efficiency. They are commonly deployed with the strategy that schedules constraint checking immediately upon context changes. This assures no missed inconsistency, but also limits the detection efficiency. One may break the limit by grouping context changes for checking together, but this can cause severe inconsistency missing problem (up to 79.2%). In this article, we propose a novel strategy GEAS to isolate latent interferences among context changes and schedule constraint checking with adaptive group sizes. This makes GEAS not only improve the detection efficiency, but also assure no missed inconsistency with theoretical guarantee. We experimentally evaluated GEAS with large-volume real-world context data. The results show that GEAS achieved significant efficiency gains for context inconsistency detection by 38.8–566.7% (or 1.4x–6.7x). When enhanced with an extended change-cancellation optimization, the gains were up to 2,755.9% (or 28.6x).

Index Terms—Context inconsistency detection, consistency constraint, scheduling strategy, susceptibility/cancellation condition.

1 Introduction

With the advances of new sensing and actuation technologies, many applications (e.g., self-driving vehicles [1], [2], [3] and mobile apps [4]) from cyber-physical, wireless, and cloud computing fields now become increasingly adaptive. They rely on *contexts* [5] to understand their running environments and make smart adaptation to better serve their users. The contexts typically refer to the values of concerned environmental attributes (e.g., location and weather for a self-driving vehicle), which can be used by applications to decide their current situations (e.g., running on a slippery road in a raining day) and best next actions (e.g., slowing down). One outstanding problem is that contexts, typically collected from noisy environments, can be easily inaccurate, incomplete or even conflicting with each other. This is known as *context inconsistency* [6], [7], which, if left unattended, can cause an application's abnormal adaptation or even failure.

To address the context inconsistency problem, one promising approach is to check contexts against predefined *consistency constraints* [8], [9] to detect context inconsistencies on behalf of applications. Then problematic contexts involved in the detected inconsistencies can be identified in time and then isolated from being accessed by applications. The constraint checking process is the kernel part of context inconsistency detection. It is typically required to be *efficient* so that timely handling of problematic contexts can be supported, e.g., discarding these contexts or fixing their problems for a self-driving vehicle application. For this purpose, various constraint checking techniques have been proposed, such as ECC [8], PCC [6], Con-C [10], and GAIN [11], based on different incremental or parallel mechanisms.

These constraint checking techniques are typically deployed with a naïve strategy, which schedules them *immediately* upon any context change. This *immediate scheduling* strategy guarantees in theory no missed context inconsistencies since it captures every possible impact caused by any context change on the consistency constraints under checking. However, it at the same time also limits the efficiency of context inconsistency detection due to its frequent scheduling of constraint checking. In practice, this could cause it impossible to examine all impacts from numerous context changes in short time when deployed to heavy-workload scenarios, making itself actually not usable in preventing missed context inconsistencies. As our later case study in the evaluation shows, when handling context changes from a heavy-workload taxi application scenario, four constraint checking techniques (i.e., ECC, PCC, Con-C, and GAIN) combined with the immediate scheduling strategy were all subject to serious inconsistency missing and wrongly-reporting problems (52.6–98.7% false negative and 10.0–60.0% false positive rates, respectively).

To address this problem, one may choose to *group* multiple consecutive context changes into one batch, and check them *together* for reducing the number of scheduled constraint checking. This *batch-based scheduling* strategy can certainly improve the efficiency of context inconsistency detection by merging constraint checking for multiple context changes into one. In our later evaluation, when setting

• *H. Wang, C.Xu, B. Guo, X. Ma and J. Lu are with the State Key Laboratory for Novel Software Technology and the Department of Computer Science and Technology, Nanjing University, Nanjing (210023), Jiangsu, China.*
E-mail: cocowhy1013@gmail.com, changxu@nju.edu.cn, bingying_nju@outlook.com, xxm@nju.edu.cn, lj@nju.edu.cn.

the batch size to 2, 4, and 6, the efficiency improvement can be 103.2%, 305.0%, and 508.7%, respectively, which is desirable. However, this strategy can also cause severe inconsistency missing problems. This is because the context changes grouped in one batch may possibly contain latent interferences among each other, which can lead to some inconsistencies caused by certain context changes becoming *undetectable* due to the presence of other context changes in the same batch. For example, with the aforementioned batch size setting (i.e., 2, 4, and 6), the inconsistency missing rate can be 51.8%, 68.9%, and 74.8%, respectively. This result is surprisingly negative, severely impacting the quality of context inconsistency detection. We shall explain its reason using an illustrative example later in Section 2.4.

To address this efficiency-quality dilemma, in this article, we propose a novel strategy, named *GEneric Adaptive Scheduling* (*GEAS*), to both improve the efficiency of context inconsistency detection and guarantee no missed context inconsistencies in the detection. GEAS automatically identifies latent interferences among consecutive context changes, and isolates them by separate batches with adaptive sizes. Then GEAS schedules constraint checking for such batches, achieving both efficiency gains and detection quality. Besides, GEAS is generic and can apply to existing constraint checking techniques (e.g., ECC [8], PCC [6] , Con-C [10], and GAIN [11]), making it practically useful. We also formally analyze or prove these properties of GEAS (i.e., efficiency gain, detection quality, and generality) later in Section 3.

Our key insight behind GEAS is that, since latent interferences from certain context changes can cause some inconsistencies undetectable, one can choose to break such grouping, i.e., avoiding letting these changes in the same batch. By doing so, one can recover such otherwise undetectable inconsistencies. We observe that only certain combinations of context changes can form such latent interferences, and therefore one can derive such combinations in advance and proactively avoid their presence in context inconsistency detection. We name such combinations *susceptibility conditions* (or *s-conditions* for short), which, once available, one can use to decide whether to check a new context change with earlier ones together (i.e., enlarging an existing batch) or with later ones together (i.e., forming a new batch). This decision essentially changes the batch size adaptively in order to avoid the presence of latent interferences among grouped context changes, thus achieving no missed context inconsistencies in the detection. For example, in our later evaluation, GEAS applied s-conditions and formed batches with adaptive sizes (3.6 on average), improving the detection efficiency and causing no missed inconsistency. As a comparison, the aforementioned batch-based scheduling caused an inconsistency missing rate of 51.8% even with its minimal batch size of two.

Given a sequence of context changes, the s-conditions form separate batches, each of which contains no latent interference among any two context changes in the same batch. Besides, each batch has been maximized in its size for the detection efficiency. This is with the premise that all context changes should be checked. Nevertheless, we observe that some context changes might be *redundant* in the sense that one change could generate certain impact right opposite to that by another change in constraint checking, e.g., deleting $a = 6$ generates an opposite impact to adding $a = 7$, for a Boolean function that returns ($a > 5$? True : False). In other words, they form an inter-cancellation relationship with each other if existing in the same batch. Therefore, one can choose not to check such change pairs as if both of them were not present. We name such relationship a *cancellation condition* (or *c-condition* for short). With c-conditions, a sequence of batches formed by s-conditions can further be optimized by that: (1) pairs of context changes that satisfy the c-conditions are removed, reducing the total number of context changes that should be checked; (2) some original batches can be now merged due to the removed changes, potentially increasing the average batch size. For example, in our later evaluation, on top of s-conditions, GEAS applied c-conditions and removed 80.1% total context changes. If counting these removed changes, GEAS actually increases its average batch size from 3.6 to 13.7, further improving its detection efficiency.

Combining our proposed s-conditions and c-conditions, we evaluate GEAS on a taxi application with large-volume real-world data. Our experimental results show that GEAS greatly improved the efficiency of context inconsistency detection, being 38.8–566.7% (or 1.4x–6.7x) of that of existing constraint checking techniques (ECC [8], PCC [6], Con-C [10], and GAIN [11]), when enabling s-conditions only. When enabling both s-conditions and c-conditions, the efficiency of GEAS-aided context inconsistency detection could be up to 2,755.9% (or 28.6x). Note that the efficiency gains were calculated with GEAS's overhead (e.g., that for condition evaluation, batch forming, and checking scheduling) counted in. Besides, these efficiency gains came with no missed context inconsistencies, validating GEAS's unique effectiveness, also incurring very small overhead. Moreover, we also evaluated GEAS using the taxi data in a case-study setting (i.e., following actual time restrictions). Besides significant efficiency improvement, GEAS also exhibited its unique superiority over other scheduling strategies on the quality of inconsistency detection results no matter which constraint checking technique is combined with. For example, when ECC is combined with the immediate scheduling strategy, it causes 94.4% false negatives and 31.0% false positives in its detection results, while with our GEAS, ECC realized perfect false negative and negative rates (both zero), and at the same time improved the detection efficiency over 1,300% (or 14.0x).

In summary, we make the following contributions in this article (second, third, and part of fourth ones are major extensions over GEAS's preliminary version [12]):

- We propose the notion of susceptibility condition, and use it to identify and isolate latent interferences among context changes by adaptive batches, thus achieving highly-efficient zero-missing context inconsistency detection.
- We propose the notion of cancellation condition, and use it to refine the batches by identifying and removing impact-opposite context changes, further improving the detection efficiency.
- We formally analyze or prove GEAS's three properties, namely, efficiency gain, detection quality, and generality.

- We evaluate GEAS's performance with large-volume real-world taxi data through both controlled experimentation and case study ways, validating its general effectiveness on improving the efficiency of context inconsistency detection and guaranteeing zero missed inconsistency.

The remainder of this article is organized as follows. Section 2 introduces background knowledge, gives a motivating example, and formulates our target problem. Section 3 proposes the notion of susceptibility condition, and based on it presents a novel scheduling strategy, GEAS, to achieve efficient context inconsistency detection with quality guarantee. Section 4 extends GEAS for further improved efficiency by a dedicated cancellation condition based optimization. Section 5 experimentally evaluates GEAS's performance with respect to existing constraint checking techniques, ECC [8], PCC [6], Con-C [10], and GAIN [11], on a real-world taxi application. Section 6 discusses related issues on the usage of GEAS in practice. Finally, Section 7 presents the related work in recent years, and Section 8 concludes this article.

2 Background and Problem Formulation

In this section, we introduce some background knowledge and define necessary concepts for subsequent discussions. Then, we present a motivating example to illustrate the efficiency-quality dilemma in context inconsistency detection. Finally, we formulate our target problem to solve for addressing the dilemma.

2.1 Background

Consistency management has been well recognized as an important research problem in the software engineering community. Typically, during the development of various software artifacts (e.g., requirement models, design models, source code, test cases, and configuration files), inconsistencies or conflicts among these artifacts can naturally arise due to collaborative or distributed development [13], thus calling for the need of automatically detecting and resolving such inconsistencies. Consistency management typically relies on constraint checking techniques, which check a set of software artifacts under checking against a set of predefined consistency constraints to see whether any artifact violates any constraint. Such violation, if detected, is considered as an inconsistency, which should be handled in time for the consistency of software artifacts in the development.

Many software artifacts have been extensively studied for consistency management, which include XML documents [8], [9], [14], UML models [15], [16], [17], data structures [18], workflows [19], and distributed source code [20]. When these artifacts are created or changed, they are checked against consistency constraints by the notion of rules in order to detect any violation (inconsistency) if present. For example, xlinkit [8] provided an XML-based running environment to ensure the consistency of distributed, heterogeneous Web documents across different resource types. Later, it was extended to support checking the consistency of general document artifacts during software development, including design, implementation, and deployment phases [14]. Another popular example is the consistency management for UML models, which are typically used in software development. For instance, ArgoUML [17] checked static UML models for consistency against annotated consistency rules, while Blanc et. al. [16] checked dynamic UML modeling processes in terms of model construction operations and reported both structural and methodological inconsistencies.

In the existing literature on consistency management, most studied software artifacts are static (e.g., XML documents) or can only change rarely or slowly (e.g., UML models). The software artifacts studied in this article (contexts) are different, which typically change frequently. This distinguishes contexts from traditional software artifacts, causing distinct challenges and deserving further research, as we analyze next with preliminary concepts for subsequent discussions.

2.2 Preliminary

Context. A *context* refers to a piece of environmental or logical information interesting to an application [6], [7], [21]. In this article, we model a context as a finite set of elements, each of which specifies a relevant part of this context. For illustration, we consider a location-aware package delivery application, which is adapted from existing work [7], [22], [23]. The application controls multiple robots to deliver packages among various warehouses. It arranges robots for delivery according to their locations (i.e., in which warehouse) and whether they are free (i.e., just finished a task). For this application, we can model the robots currently in a specific warehouse x by a context $C_{\mathsf{x}} = \{\mathsf{r}_1, \mathsf{r}_2, \ldots\}$. Each element r_i identifies each individual robot currently in warehouse x.

Context change. A *context change* refers to any change relating to elements in a specific context. There are three types of context change, namely, *addition change* (i.e., adding a new element into a context), *deletion change* (i.e., deleting an existing element from a context), and *update change* (i.e., updating an element's value). For ease of presentation, we use "$+$", "$-$", and "$\#$" to represent the three types, respectively. For the package delivery application, if robot r_1 enters warehouse x, it will trigger an addition change, which can be represented as $<+, C_{\mathsf{x}}, \mathsf{r}_1>$; if robot r_1 leaves warehouse y, the corresponding change is a deletion one, represented as $<-, C_{\mathsf{y}}, \mathsf{r}_1>$; if r_1 just finishes a delivery task in warehouse z with its status changed to "free", it will trigger an update change, represented as $<\#, C_{\mathsf{z}}, \mathsf{r}_1, \mathsf{r}'_1>$.

Context pool. We conceptually assume the availability of a *context pool*, which collects all contexts interesting to an application, e.g., three warehouse-robot contexts (C_{x}, C_{y}, and C_{z}) for the preceding application. Many existing middleware infrastructures or frameworks [4], [24], [25], [26] support such pool-alike data structures for context-aware applications. Maintaining such a context pool helps applications to access their interesting contexts when necessary and apply collected context changes to corresponding contexts.

Consistency constraint. For the package delivery application, we use radio frequency identification (RFID) technology [27] to track when a robot enters or leaves a specific warehouse. Due to the missing or cross read [28], [29], [30],

[31] and asynchronous read (async read) [7], [32] problems with RFID data, the derived warehouse-robot contexts can contain incomplete or inaccurate data, causing context inconsistency problems. To detect such inconsistencies, we formulate *consistency constraints* [8], [9] for specifying necessary properties that must hold about the contexts. One may specify consistency constraints using a first order logic (FOL) based constraint language [6], [7] as follows:

$$\begin{aligned} f :=& \forall v \in C\ (f) \mid \exists v \in C\ (f) \mid \\ & (f)\ \mathsf{and}\ (f) \mid (f)\ \mathsf{or}\ (f) \mid (f)\ \mathsf{implies}\ (f) \mid \mathsf{not}\ (f) \mid \\ & bfunc(v_1, v_2, \ldots, v_n) \mid \mathsf{True} \mid \mathsf{False}. \end{aligned}$$

Here, C represents a context from an application's context pool; v_i is a variable, which takes an element from a context as its value; terminal $bfunc$ is a domain-specific function, which takes values of variables as input and returns True or False. Consistency constraints are typically formulated from physical laws or application-specific requirements [6], [8], [9]. For example, regarding our preceding application, one may formulate a consistency constraint $\mathsf{S_{loc}}$ like "*no robot can stay in two warehouses* x *and* y *at the same time*" as follows:

$$\mathsf{S_{loc}} : \forall v_\mathsf{x} \in C_\mathsf{x}\ (\mathsf{not}\ (\exists v_\mathsf{y} \in C_\mathsf{y}\ (\mathsf{same}(v_\mathsf{x},\ v_\mathsf{y})))).$$

Constraint checking. With consistency constraints formulated, contexts can then be validated by checking them against the constraints to see whether there is any violation. If yes, the violated constraint is evaluated to False (a.k.a., obtaining a *truth value* of False), and returns a set of *links* [8], representing detected context inconsistencies. Each link explains which particular elements in concerned contexts cause the constraint's violation. This process is named *constraint checking* [6], [9], [10], [11], which is the kernel part of inconsistency detection for software artifacts.

Consider the preceding application. Suppose that its context pool contains two contexts, namely, $C_\mathsf{x} = \{\mathsf{r_1}, \mathsf{r_2}\}$ and $C_\mathsf{y} = \{\mathsf{r_3}, \mathsf{r_4}\}$. Now robot $\mathsf{r_1}$ leaves warehouse x and enters warehouse y. This movement triggers two changes, i.e., $<-, C_\mathsf{x}, \mathsf{r_1}>$ and $<+, C_\mathsf{y}, \mathsf{r_1}>$. When applying them, the application's contexts become $C_\mathsf{x} = \{\mathsf{r_2}\}$ and $C_\mathsf{y} = \{\mathsf{r_1}, \mathsf{r_3}, \mathsf{r_4}\}$. Now one schedules constraint checking for the current contexts against consistency constraint $\mathsf{S_{loc}}$. Then $\mathsf{S_{loc}}$ would be evaluated to True and there is no context inconsistency. However, if the change of $\mathsf{r_1}$ leaving warehouse x is lost due to a missing RFID read, $\mathsf{r_1}$ would be conjectured still in warehouse x although it has entered warehouse y. Now the corresponding contexts are $C_\mathsf{x} = \{\mathsf{r_1}, \mathsf{r_2}\}$ and $C_\mathsf{y} = \{\mathsf{r_1}, \mathsf{r_3}, \mathsf{r_4}\}$. Checking the contexts against constraint $\mathsf{S_{loc}}$ would report a context inconsistency by a link containing $\mathsf{r_1}$ as the value for both variables v_x and v_y, suggesting that robot $\mathsf{r_1}$ appears in two warehouses at the same time. Formally, such a link is represented as $(\mathsf{violated}, \{(v_\mathsf{x}, \mathsf{r_1}), (v_\mathsf{y}, \mathsf{r_1})\})$, in which "violated" means that the concerned constraint has been violated and the part within the pair of braces "{}" specifies the assignment of value $\mathsf{r_1}$ to both variables v_x and v_y. We note that our subsequent discussions will not cover the internal details of such links, and here this link is for example and illustration only.

2.3 Scheduling Strategy

Typically, constraint checking is scheduled immediately upon each collected context change. This is known as the *immediate scheduling* strategy. Existing constraint checking techniques (ECC [8], PCC [6], Con-C [10], and GAIN [11]) typically use immediate scheduling for timely detection of context inconsistency. For the aforementioned two context changes, $<-, C_\mathsf{x}, \mathsf{r_1}>$ and $<+, C_\mathsf{y}, \mathsf{r_1}>$, immediate scheduling would schedule constraint checking twice, one for each change, according to their order. Thus, if there is any inconsistency that could be caused by any change, immediate scheduling can detect it in time. However, this strategy also limits the efficiency of context inconsistency detection when context changes are very frequent. Therefore, immediate scheduling can hardly suit heavy-workload application scenarios.

When context changes are frequent, existing constraint checking techniques can also choose to check collected changes by grouping. This strategy is known as *batch-based scheduling*. For the aforementioned two context changes, $<-, C_\mathsf{x}, \mathsf{r_1}>$ and $<+, C_\mathsf{y}, \mathsf{r_1}>$, batch-based scheduling can schedule constraint checking only once by taking the two changes as a whole (i.e., batch size of two). If there is no context inconsistency that can be caused by the first change, this treatment can reduce constraint checking once without any negative consequence.

2.4 Motivating Example

Regarding the preceding two context changes, it is fortunate that applying them *together* to the context pool and scheduling constraint checking only *once* will not miss any context inconsistency. However, it may not always be the case. In the following, we modify these context changes to illustrate what could be negative consequences of such batch-based scheduling.

Consider three robots, $\mathsf{r_1}$, $\mathsf{r_2}$, and $\mathsf{r_3}$, that deliver packages among two warehouses, x and y. Initially, no robot stays in any warehouse. Then, the three robots move as follows: $\mathsf{r_1}$ enters x and then leaves x; $\mathsf{r_2}$ enters x, leaves x, and then enters y; $\mathsf{r_3}$ enters y, leaves y, and then enters x. Unfortunately, some cross, missing, and async RFID reads occur during this process: $\mathsf{r_1}$ is detected to enter y when it is still in x (cross read); the read is lost when $\mathsf{r_2}$ leaves x (missing read); the reads of $\mathsf{r_3}$ leaving y and entering x are switched in order (async read). These robot movements and occurring cross, missing, and async reads together lead to eight context changes, as illustrated in Fig. 1.

Suppose that one uses the preceding constraint $\mathsf{S_{loc}}$ for governing the consistency of contexts. When applying these context changes and scheduling constraint checking upon each change (i.e., immediate scheduling), one can obtain three inconsistencies, $\mathsf{inc_1}$, $\mathsf{inc_2}$, and $\mathsf{inc_3}$, at $\mathsf{chg_3}$, $\mathsf{chg_6}$, and $\mathsf{chg_7}$, respectively, as illustrated in Fig. 1. This result is desirable, as the detected context inconsistencies are complete for this example. However, real-world package delivery applications like the Amazon Kiva system [33], [34] are much more complex. They can control hundreds or even thousands of robots across numerous warehouse regions at the same time. This causes an extremely high context change rate (e.g., up to 500 changes per second according to the

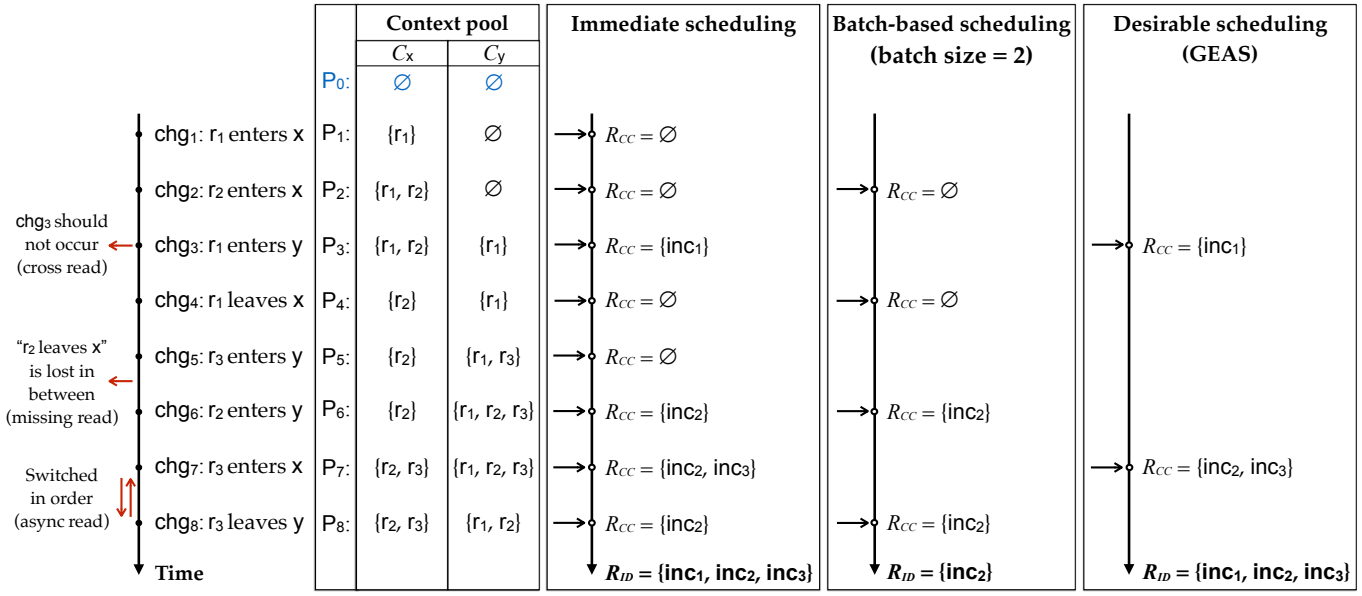


Fig. 1: Illustrative scenario – inconsistency missing problem with batch-based scheduling.

news on the STO express sorting system [35], [36] in China). For such frequent context changes, immediate scheduling of constraint checking can hardly apply.

To reduce the number of scheduled constraint checking, one can choose batch-based scheduling. For example, we set the batch size to two, and thus only need to schedule constraint checking *four* times, i.e., at chg_2, chg_4, chg_6, and chg_8, respectively, as illustrated in Fig. 1. This treatment dramatically reduces the number of constraint checking by *half* (i.e., almost 100% efficiency improvement). However, this also leads to only inc_2 being detected, with inc_1 and inc_3 missed (i.e., 66.7% missing rate).

To address this efficiency-quality dilemma, a desirable strategy should schedule constraint checking only *twice*, at chg_3 and chg_7, respectively. This treatment can correctly detect all inconsistencies (i.e., zero missing rate) and at the same time greatly improve the efficiency (i.e., almost 300%). Nevertheless, how can one achieve such a desirable strategy? In the following, we formulate this problem and discuss how to solve it for achieving this desirable strategy.

2.5 Problem Formulation

To formulate our problem, we first explain how a context pool P evolves with collected context changes. Suppose that the whole process starts at time point t_0 and each context change is collected at a distinct time point after t_0 (i.e., chg_i is collected at time point t_i). We use P_i to represent the updated context pool at time point t_i after applying chg_i to the contexts in the pool (let P_0 be the initial context pool). Then, $P_{i+1} = \mathsf{apply}(P_i, chg_{i+1})$.

We then define two concepts, R_{CC} (*constraint checking result*) and R_{ID} (*inconsistency detection result*). The former $R_{CC}(s, P, t)$ represents the constraint checking result of the contexts in pool P at time point t against consistency constraint s, which is a set of detected context inconsistencies for this particular checking. For example, in immediate scheduling, the R_{CC} value is $\varnothing$ when checking P at chg_2 (i.e., P_2) against $\mathsf{S}_{\mathsf{loc}}$, and is $\{\mathsf{inc}_2\}$ for P_6, as illustrated in Fig. 1. The latter $R_{ID}(s, P, t)$ represents the inconsistency detection result, which is the set of all context inconsistencies ever detected for the contexts in pool P by time point t against consistency constraint s. For example, in immediate scheduling, the R_{ID} value is $\{\mathsf{inc}_1\}$ for $t = 4$ and $\{\mathsf{inc}_1, \mathsf{inc}_2, \mathsf{inc}_3\}$ for $t = 8$.

We note that: (1) R_{CC} and R_{ID} values are related with each other, and (2) they also rely on the scheduling strategy used for constraint checking. We formulate such relationships as follows. Given a scheduling strategy, let its associated time points for scheduling constraint checking be a set: $TP = \{t_{i_1}, t_{i_2}, \ldots, t_{i_m}\}$. Then $R_{ID}(s, P, t)$ is the union of all $R_{CC}(s, P, t_{i_k})$ values for all t_{i_k} from TP, i.e.,

$$R_{ID}(s, P, t) = \cup_{k=1}^{m} R_{CC}(s, P, t_{i_k}).$$

Consider the example in Fig. 1. If we use immediate scheduling, then its TP is $\{\mathsf{t}_1, \mathsf{t}_2, \ldots, \mathsf{t}_8\}$ and corresponding $R_{ID}(s, P, t)$ is:

$$= \cup_{k=1}^{8} R_{CC}(\mathsf{S}_{\mathsf{loc}}, P, \mathsf{t}_k) = \{\mathsf{inc}_1, \mathsf{inc}_2, \mathsf{inc}_3\}.$$

If we use batch-based scheduling, its TP is $\{\mathsf{t}_2, \mathsf{t}_4, \mathsf{t}_6, \mathsf{t}_8\}$ and $R_{ID}(s, P, t)$ is:

$$= \cup_{k=1}^{4} R_{CC}(\mathsf{S}_{\mathsf{loc}}, P, \mathsf{t}_{2k}) = \{\mathsf{inc}_2\}.$$

From this formulation and calculation, one can understand why batch-based scheduling would miss context inconsistencies in the detection. Thus, even if it improves the detection efficiency, batch-based scheduling is undesirable.

Then, to achieve our aforementioned desirable strategy, i.e., scheduling only twice but detecting all three context inconsistencies, we formulate its objectives as follows. Given a sequence of context changes, chg_1, chg_2, …, and chg_n, which are collected at time points t_1, t_2, …, and t_n, let its TP be $\{t_{i_1}, t_{i_2}, \ldots, t_{i_m}\}$ $(1 \leq i_1 < i_2 < \ldots < i_m \leq n)$.

(1) Quality objective. The scheduling strategy should not miss any context inconsistency, i.e., its R_{ID} value should always be equal to that of immediate scheduling. Formally,

$$\cup_{k=1}^{m} R_{CC-des}(s, P, t_{i_k}) = \cup_{k=1}^{n} R_{CC-imd}(s, P, t_k).$$

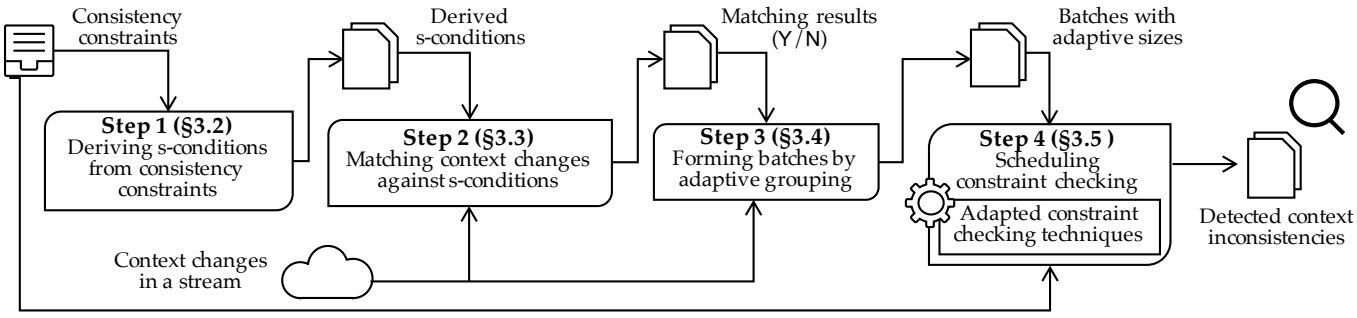


Fig. 2: GEAS overview.

(2) Efficiency objective. The scheduling strategy should work more efficiently than immediate scheduling, i.e., it does not have to schedule constraint checking for each context change. Since it checks context changes by grouping, its efficiency improvement can be *roughly* estimated as (assuming that each checking takes the same time):

$$(n - m) \,/\, m.$$

For the example in Fig. 1, the desirable strategy's efficiency improvement over immediate scheduling is $(8 - 2) / 2 = 300\%$. Our later evaluation would calculate the efficiency improvement by actual time cost, which is proportional to this rough estimation.

In the following, we propose our GEAS strategy to achieve this desirable strategy, and make its efficiency improvement as high as possible. By doing so, we address the efficiency-quality dilemma in context inconsistency detection.

3 Generic Adaptive Scheduling

In this section, we elaborate on our GEAS strategy and explain how to apply it to existing constraint checking techniques (ECC [8], PCC [6], Con-C [10], and GAIN [11]) for efficient context inconsistency detection.

3.1 Overview

We give the GEAS overview in Fig. 2. It consists of four steps. In the first step (Section 3.2), GEAS derives susceptibility conditions (s-conditions) from consistency constraints statically. In the second step (Section 3.3), it matches collected context changes against these derived s-conditions at runtime. In the third step (Section 3.4), it forms batches with adaptive sizes according to the matching results. Finally, in the last step (Section 3.5), it schedules constraint checking for the formed batches and reports detected context inconsistencies. We would also explain in this step how to adapt existing constraint checking techniques such that they can work with GEAS for efficient context inconsistency detection.

We use the preceding example in Fig. 1 to illustrate this process. For consistency constraint $\mathsf{S}_{\mathsf{loc}}$, GEAS derives 16 s-conditions. Then, during the context inconsistency detection, the first three context changes do not match any s-condition with each other, and so they form a batch (chg_1, chg_2, and chg_3). When the fourth change comes, it matches an s-condition with chg_1) (enter-leave interference, to be explained later), and so it is put into the next batch. Similarly, the next three changes (chg_5, chg_6, and chg_7) do not match any s-condition with chg_4 and each other, and so they are put into the same batch as chg_4. The last change, chg_8, matches an s-condition with chg_5 (enter-leave interference), and so it goes to the next new batch. Since the third batch is not closed yet, GEAS schedules constraint checking twice for the first two batches, respectively, at chg_3 and chg_7. It thus obtains two results, $\{\mathsf{inc}_1\}$ and $\{\mathsf{inc}_2, \mathsf{inc}_3\}$. After merging them, the final detection result R_{ID} is $\{\mathsf{inc}_1, \mathsf{inc}_2, \mathsf{inc}_3\}$, which is the desirable one.

We owe this ability of forming adaptive batches and isolating change interferences to GEAS's derived s-conditions. In the following, we explain how to systematically derive s-conditions from given consistency constraints.

3.2 Step 1: Deriving Susceptibility Conditions from Consistency Constraints

GEAS uses s-conditions to model the interferences among context changes that can cause context inconsistencies undetectable. To derive s-conditions, we need to analyze the impact that can be caused by a certain context change on a consistency constraint. We first introduce two impact types, *inc+ impact* and *inc− impact*. If any new context change can cause a consistency constraint to change its original truth value from True to False, the change can potentially cause new context inconsistencies. In this case, the impact caused by this context change belongs to *inc+ impact*. On the contrary, if any context change can cause the constraint to change its truth value from False to True, the change can potentially cause existing context inconsistencies undetectable. Similarly, the impact caused by this context change belongs to *inc− impact*. Note that a context change can cause both inc+ and inc− impacts.

According to what impact type(s) a context change can cause, we classify context changes into three types:

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

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

Definition 3 (inc? change). *If a context change can cause both inc+ and inc− impacts, it is classified as an inc? change,*

TABLE 1: Deduction rules for $Set_{inc+}(s)$, $Set_{inc-}(s)$, and $Set_{inc?}(s)$.

Formula type	Deduction rules		
	$Set_{inc+}(f)$	$Set_{inc-}(f)$	$Set_{inc?}(f)$
$f = \forall v \in C\ (f_1)$	$Set_{inc+}(f_1)\ \cup\ \{<+, C>\}$	$Set_{inc-}(f_1) \cup \{<-, C>\}$	$Set_{inc?}(f_1) \cup \{<\#, C>\}$
$f = \exists v \in C\ (f_1)$	$Set_{inc+}(f_1)\ \cup\ \{<-, C>\}$	$Set_{inc-}(f_1) \cup \{<+, C>\}$	$Set_{inc?}(f_1) \cup \{<\#, C>\}$
$f = (f_1)$ and (f_2)	$Set_{inc+}(f_1) \cup Set_{inc+}(f_2)$	$Set_{inc-}(f_1) \cup Set_{inc-}(f_2)$	$Set_{inc?}(f_1) \cup Set_{inc?}(f_2)$
$f = (f_1)$ or (f_2)	$Set_{inc+}(f_1) \cup Set_{inc+}(f_2)$	$Set_{inc-}(f_1) \cup Set_{inc-}(f_2)$	$Set_{inc?}(f_1) \cup Set_{inc?}(f_2)$
$f = (f_1)$ implies (f_2)	$Set_{inc-}(f_1) \cup Set_{inc+}(f_2)$	$Set_{inc+}(f_1) \cup Set_{inc-}(f_2)$	$Set_{inc?}(f_1) \cup Set_{inc?}(f_2)$
$f =$ not (f_1)	$Set_{inc-}(f_1)$	$Set_{inc+}(f_1)$	$Set_{inc?}(f_1)$
$f = bfunc(v_1, v_2, ...)$	$\varnothing$	$\varnothing$	$\varnothing$

indicating that this context change can potentially cause new inconsistencies, existing inconsistencies undetectable, or both.

We give several examples on classifying context changes into the preceding three types according to their caused impact types. For ease of presentation, we represent a context change by a simpler form by removing its element information. For example, the three preceding context changes $<+, C_{\mathsf{x}}, \mathsf{r}_1>$, $<-, C_{\mathsf{y}}, \mathsf{r}_1>$, and $<\#, C_{\mathsf{z}}, \mathsf{r}_1, \mathsf{r}'_1>$ are now represented as $<+, C_{\mathsf{x}}>$, $<-, C_{\mathsf{y}}>$, and $<\#, C_{\mathsf{z}}>$. Consider a consistency constraint s with a universal formula like $\forall v \in C\ (bfunc(v))$ and three context changes $<+, C>$, $<-, C>$, $<\#, C>$. For this constraint, $<+, C>$ is an inc+ change because adding an element into C can possibly cause constraint s to change its truth value from True to False, and meanwhile it can never cause s to change its truth value from False to True. Similarly, $<-, C>$ is an inc− change because deleting an existing element from C can possibly cause constraint s to change its truth value from False to True, and meanwhile it can never cause s to change its truth value from True to False. $<\#, C>$ is an inc? change because updating an element in C can change the element's value arbitrarily, and thus it might cause unpredictable impact.

As mentioned earlier, we derive s-conditions from consistency constraints, so that one can match context changes against these conditions to form batches for scheduling (Fig. 2). The derivation of s-conditions can be done statically, as long as the constraints are available. The basic idea is to: (1) first examine all possible context changes for a consistency constraint, and classify them into three types, namely, inc+, inc−, and inc? changes, and (2) then compose s-conditions for the constraint from these classified inc+, inc−, and inc? changes.

Step 1.1: Classifying impact types. We classify all possible context changes for a consistency constraint s into three sets, namely, $Set_{inc+}(s)$, $Set_{inc-}(s)$, and $Set_{inc?}(s)$, which contain inc+, inc−, and inc? changes, respectively. We *recursively* deduct the three sets according to the formula types used in a constraint. Table 1 gives all deduction rules. For example, consider formula $f = \forall v \in C\ (f_1)$. The elements in formula f's inc+ change set $Set_{inc+}(f)$ come from its subformula f_1's inc+ change set $Set_{inc+}(f_1)$, plus its newly introduced $<+, C>$, as explained earlier.

We apply the deduction rules in Table 1 to the preceding consistency constraint $\mathsf{S}_{\mathsf{loc}}$, and deduct its three change sets, $Set_{inc+}(\mathsf{S}_{\mathsf{loc}})$, $Set_{inc-}(\mathsf{S}_{\mathsf{loc}})$, and $Set_{inc?}(\mathsf{S}_{\mathsf{loc}})$, as follows:

$$
\begin{aligned}
&Set_{inc+}(\mathsf{S}_{\mathsf{loc}}) \\
&= Set_{inc+}(\forall v_{\mathsf{x}} \in C_{\mathsf{x}}\ (\mathsf{not}\ (\exists v_{\mathsf{y}} \in C_{\mathsf{y}}\ (\mathsf{same}(v_{\mathsf{x}},\ v_{\mathsf{y}}))))) \\
&= Set_{inc+}(\mathsf{not}\ (\exists v_{\mathsf{y}} \in C_{\mathsf{y}}\ (\mathsf{same}(v_{\mathsf{x}},\ v_{\mathsf{y}})))) \cup \{<+, C_{\mathsf{x}}>\} \\
&= Set_{inc-}(\exists v_{\mathsf{y}} \in C_{\mathsf{y}}\ (\mathsf{same}(v_{\mathsf{x}},\ v_{\mathsf{y}}))) \cup \{<+, C_{\mathsf{x}}>\} \\
&= Set_{inc-}(\mathsf{same}(v_{\mathsf{x}},\ v_{\mathsf{y}})) \cup \{<+, C_{\mathsf{y}}>\} \cup \{<+, C_{\mathsf{x}}>\} \\
&= \{<+, C_{\mathsf{x}}>, <+, C_{\mathsf{y}}>\}.
\end{aligned}
$$

Similarly, $Set_{inc-}(\mathsf{S}_{\mathsf{loc}})$ and $Set_{inc?}(\mathsf{S}_{\mathsf{loc}})$ are:

$$
\begin{aligned}
Set_{inc-}(\mathsf{S}_{\mathsf{loc}}) &= \{<-, C_{\mathsf{x}}>, <-, C_{\mathsf{y}}>\}, \\
Set_{inc\#}(\mathsf{S}_{\mathsf{loc}}) &= \{<\#, C_{\mathsf{x}}>, <\#, C_{\mathsf{y}}>\}.
\end{aligned}
$$

In the following, we explain how we obtain the deduction rules in Table 1 by a theorem and its proof.

Theorem 1 (Completeness and soundness of the deduction rules). *Given a consistency constraint s, the deduction rules in Table 1 generate three complete and sound change sets, $Set_{inc+}(s)$, $Set_{inc-}(s)$, and $Set_{inc?}(s)$, which include all and correct inc+, inc−, and inc? changes for this constraint, respectively.* □

Proof. We use induction to prove the completeness and soundness of the deduction rules. Note that the completeness and soundness here are with respect to generating the elements in the three context change sets, i.e., all and correct inc+, inc−, and inc? changes can be generated for the three sets, respectively, by the deduction rules. Consider a consistency constraint s. Let its height be h (i.e., maximal number of recursion depth in any part of this constraint, e.g., $\forall v \in C\ (f)$ has a height of one).

(1) Base case. When h = 1, constraint s must take the form of $\forall v \in C\ (bfunc(v))$ or $\exists v \in C\ (bfunc(v))$. Due to similarity in the proof, we consider only the former.

For constraint s, its all possible context changes relate to its universal formula part only, which are $<+, C>$, $<-, C>$, and $<\#, C>$. We examine them in turn: (1) $<+, C>$ can cause s's truth value to change only in three ways: True → False, False → False, and True → True, but never False to True, and thus it belongs to $Set_{inc+}(s)$; (2) similarly, $<-, C>$ belongs to $Set_{inc-}(s)$; (3) $<\#, C>$ can cause s's truth value to change in all four ways, and thus it belongs to $Set_{inc?}(s)$.

Our deduction rules in Table 1 exactly give this result since constraint s's $bfunc$ part does not relate to any context change for constraint s:

$$\begin{aligned} Set_{inc+}(s) &= Set_{inc+}(\forall v \in C\ (bfunc(v))) \\ &= Set_{inc+}(bfunc(v)) \cup \{<+, C>\} \\ &= \varnothing \cup \{<+, C>\} \\ &= \{<+, C>\}. \end{aligned}$$

Similarly, $Set_{inc-}(s)$ = {<−, C>} and $Set_{inc?}(s)$ = {<#, C>}.

Therefore, when h = 1, the completeness and soundness hold for the deduction rules.

(2) Induction step. Still let constraint s's height be h. Suppose that when $1 \leq h \leq k$, the three change sets by the deduction rules for s are complete and sound. We now consider $h = k + 1$. In this case, constraint s can only take one of the six forms: $\forall v \in C\ (f_1)$, $\exists v \in C\ (f_1)$, (f_1) **and** (f_2), (f_1) **or** (f_2), (f_1) **implies** (f_2), and **not** (f_1), since $bfunc$ does not incur any recursion. Due to similarity in the proof, we consider only $\forall v \in C\ (f_1)$, (f_1) **and** (f_2), and **not** (f_1).

Case $s = \forall v \in C\ (f_1)$. Let subformula f_1's height be h_{f_1}, which must be k. Due to the induction assumption, f_1's three sets, $Set_{inc+}(s)$, $Set_{inc-}(s)$, and $Set_{inc?}(s)$, include all and correct inc+, inc−, and inc? changes for f_1, respectively. Due to similarity in the proof, we prove the completeness and soundness for $Set_{inc+}(s)$ only. The set should contain changes from subformula f_1 and the universal formula itself. For the latter, it is trivial as only {<+, C>} from its three changes can cause inc+ impact but never inc− impact, and thus goes to $Set_{inc+}(s)$, as explained in the basic step. For the former, according to the semantics of the universal operator, any change that can cause inc+ impact but never inc− impact to f_1 also does so to s, and thus goes to $Set_{inc+}(s)$ as well. Meanwhile, any other change that can cause inc− impact to f_1 also does so to s, and thus should not go to $Set_{inc+}(s)$. Combining the two parts together, we obtain the deduction rule $Set_{inc+}(f_1) \cup$ {<+, C>} for $Set_{inc+}(s)$.

Case $s = (f_1)$ and (f_2). Let subformula f_1's height be h_{f_1} and f_2's height be h_{f_2}. According to the definition of height, at least one of h_{f_1} and h_{f_2} is k. Without the loss of generality, we assume $h_{f_1} = k$ and $h_{f_2} = k - 1$. Due to the induction assumption, f_1's and f_2's change sets include all and correct inc+, inc−, and inc? changes for f_1 and f_2, respectively. Due to similarity in the proof, we prove the completeness and soundness for $Set_{inc+}(s)$ only. The set should contain changes from subformulas f_1 and f_2. For the former, according to the semantics of the **and** operator, any change that can cause inc+ impact but never inc− impact to f_1 also does so to s, and thus goes to $Set_{inc+}(s)$. Meanwhile, any other change that can cause inc− impact to f_1 also does so to s, and thus should not go to $Set_{inc+}(s)$. For the latter, it can be proved similarly that for f_2, only changes from $Set_{inc+}(f_2)$ should go to $Set_{inc+}(s)$. Combining the two parts together, we obtain the deduction rule $Set_{inc+}(f_1) \cup Set_{inc+}(f_2)$ for $Set_{inc+}(s)$.

Case $s =$ not (f_1). Let subformula f_1's height be h_{f_1}, which must be k. Due to the induction assumption, f_1's three sets, $Set_{inc+}(s)$, $Set_{inc-}(s)$, and $Set_{inc?}(s)$, include all and correct inc+, inc−, and inc? changes for f_1, respectively. Due to similarity in the proof, we prove the completeness and soundness for $Set_{inc+}(s)$ only. The set should contain changes only from subformula f_1 since the **not** operator itself does not introduce any context change. For subformula f_1, according to the semantics of the **not** operator, any change that can cause inc− impact but never inc+ impact to f_1 can cause inc+ impact but never inc− impact to s, and thus goes to $Set_{inc+}(s)$ as well. Meanwhile, any other change that can cause inc+ impact to f_1 can possibly cause inc− impact to s, and thus should not go to $Set_{inc+}(s)$. Combining the two parts together, we obtain the deduction rule $Set_{inc-}(f_1)$ for $Set_{inc+}(s)$.

Therefore, when $h = k + 1$, the completeness and soundness also hold for the deduction rules.

As a summary, combining the basic and induction steps, the deduction rules in Table 1 always generate three complete and sound change sets, $Set_{inc+}(s)$, $Set_{inc-}(s)$, and $Set_{inc?}(s)$, which include all and correct inc+, inc−, and inc? changes for any given consistency constraint s. □

Note that the proof of Theorem 1 also explains how we obtain the deduction rules. With these rules, one can deduct three change sets, $Set_{inc+}(s)$, $Set_{inc-}(s)$, and $Set_{inc?}(s)$, for any consistency constraint s. Based on the three change sets, one can further compose s-conditions for constraint s.

Step 1.2: Composing s-conditions. As mentioned earlier, s-conditions model the latent interferences among context changes, so that one can use them to prevent context inconsistencies from being undetectable. The interferences actually describe a pair of *opposite* impacts, one incurring inconsistency while another causing it undetectable. Thus the first one concerns inc+ impact, which comes from Set_{inc+} or $Set_{inc?}$ change set, and the second one concerns inc− impact, which comes from Set_{inc-} or $Set_{inc?}$ change set. Therefore, we define s-condition as follows:

Definition 4 (Susceptibility condition or s-condition). *Given a consistency constraint s, its s-conditions are formed by such pairs, the first part of which comes from its $Set_{inc+}(s)$ or $Set_{inc?}(s)$ change set, and the second part comes from its $Set_{inc-}(s)$ or $Set_{inc?}(s)$ change set.*

According to the definition, an s-condition can take only one of four forms, namely, (inc+, inc−), (inc+, inc?), (inc?, inc−), and (inc?, inc?). For example, consider the preceding consistency constraint $\mathsf{S_{loc}}$. According to its deducted three change sets, $Set_{inc+}(\mathsf{S_{loc}})$, $Set_{inc-}(\mathsf{S_{loc}})$, and $Set_{inc?}(\mathsf{S_{loc}})$, in Step 1.1, one can compose its 16 s-conditions as follows (four forms, with four each form; "/" represents two possibilities for compactness):

$$\begin{aligned} &(\text{inc+, inc−}) \text{ form:} \quad (<+, C_{\mathsf{x}}/C_{\mathsf{y}}>, <-, C_{\mathsf{x}}/C_{\mathsf{y}}>), \\ &(\text{inc+, inc?}) \text{ form:} \quad (<+, C_{\mathsf{x}}/C_{\mathsf{y}}>, <\#, C_{\mathsf{x}}/C_{\mathsf{y}}>), \\ &(\text{inc?, inc−}) \text{ form:} \quad (<\#, C_{\mathsf{x}}/C_{\mathsf{y}}>, <-, C_{\mathsf{x}}/C_{\mathsf{y}}>), \\ &(\text{inc?, inc?}) \text{ form:} \quad (<\#, C_{\mathsf{x}}/C_{\mathsf{y}}>, <\#, C_{\mathsf{x}}/C_{\mathsf{y}}>). \end{aligned}$$

Let us dig a little bit more about why s-conditions can help prevent context inconsistencies from being undetectable. Let the context pool at time point t_0 be P_0, and its corresponding constraint checking result be $R_{CC}(s, P, t_0)$. Let the next two time points be t_a and t_b. Then their corresponding context changes are chg_a and chg_b, and updated

context pools are P_a and P_b, respectively. Suppose that one considers constraint s for governing the consistency of contexts in the pool, and the two changes, P_a and P_b, as a pair happen to match one of s's s-conditions (we explain the matching process later in Section 3.3). According to the definition of s-condition, this matching indicates that chg_a can cause new context inconsistency and chg_b can cause existing inconsistency undetectable. That is, we can have the following two inequations:

$$R_{CC}(s, P, t_a) - R_{CC}(s, P, t_0) \neq \varnothing,$$
$$R_{CC}(s, P, t_a) - R_{CC}(s, P, t_b) \neq \varnothing.$$

Let the left side be R_1 and R_2 for the two inequations, respectively. We note that R_1 and R_2 results may overlap (e.g., when $R_{CC}(s,\ P,\ t_0) = \{\mathsf{inc}_1\}$, $R_{CC}(s,\ P,\ t_a) = \{\mathsf{inc}_1, \mathsf{inc}_2\}$, and $R_{CC}(s,\ P,\ t_b) = \varnothing$). When this is the case, checking the two changes, chg_a and chg_b, together would cause the new inconsistency (inc_2) due to chg_a to be missed (since $R_{CC}(s,\ P,\ t_a) = \{\mathsf{inc}_1, \mathsf{inc}_2\}$ is no longer available). On the other hand, if one uses s-conditions to decide to check the two changes in different batches, the new inconsistency due to chg_a would be individually counted into the final result, i.e., not missed. Therefore, s-conditions help prevent context inconsistencies from being undetectable. We note that this is achieved in a conservative way, i.e., s-conditions try to avoid every possibility that can cause any missed inconsistency.

As a summary, in this step GEAS analyzes and derives s-conditions from consistency constraints. The s-conditions are used for matching context changes at runtime. We have explained simple cases of matching a pair of consecutive context changes against a constraint's s-conditions. In the following, we discuss general cases on how to match a sequence of context changes.

3.3 Step 2: Matching Context Changes against Susceptibility Conditions

When matching a sequence of context changes against a consistency constraint's s-conditions, GEAS needs to examine every possible pair formed by these changes. For the efficiency concern, GEAS examines the pairs in an *incremental* way.

We use the example in Fig. 1 to explain how GEAS incrementally examines context change pairs. This is supported by the notion of *batch*. The first three context changes, chg_1, chg_2, and chg_3, are all addition changes. According to our derived 16 s-conditions for constraint $\mathsf{S}_{\mathsf{loc}}$, they match no s-condition, and thus they form the initial batch. Suppose now one collects the fourth context change chg_4, which is a deletion change. Instead of examining all possible pairs formed among chg_1, chg_2, chg_3, and chg_4 (i.e., 6 chronological permutations), GEAS only examines three pairs, (chg_1, chg_4), (chg_2, chg_4), and (chg_3, chg_4), in turn. GEAS finds that the first pair already matches an s-condition ($<+, C_{\mathsf{x}}>$, $<-, C_{\mathsf{x}}>$). So GEAS stops examining the following two pairs, as this result is enough for GEAS to decide to form a new batch for chg_4 (we explain the batch-forming process later in Section 3.4).

We present this matching algorithm in Algorithm 1. For consistency constraint s, consider a new context change chg_{new}. The change is attached to each existing change in constraint s's batch in turn (Line 2) to examine whether such formed pair matches any s's s-condition (Line 4). If any matching is found, the examination terminates immediately for efficiency. Finally, the algorithm returns the matching result for chg_{new} with respect to constraint s.

Algorithm 1: (s-condition-matching) Matching context changes against s-conditions

Input: consistency constraint s, new context change chg_{new}
Output: matching result rs

```
1 rs ← False
2 for chg_i ∈ s.batch chronologically do
3     if chg_i is an inc+ or inc? change then
4         if (chg_i, chg_new) ∈ s.s-conditions then
5             rs ← True
6             break
7 return rs
```

Algorithm 2: (batch-forming) Forming batches by adaptive grouping

Input: set of consistency constraints $\mathcal{S}$, new context change chg_{new}
Output: set of consistency constraints $\mathcal{S}$ (updated)

```
1 for each s ∈ S do
2     if s-condition-matching(s, chg_new) then
3         s.newBatch ← <chg_new>
4     else
5         s.batch ← append(s.batch, chg_new)
6 return S
```

As a summary, in this step GEAS examines each pair formed by context changes against a consistency constraint's s-conditions to decide whether to form a new batch for this constraint. We explain the batch-forming process next.

3.4 Step 3: Forming Batches by Adaptive Grouping

Based on the matching results from the last step, GEAS forms batches by adaptive grouping of isolated context changes. By "adaptive", the size of each formed batch is no longer fixed, as contrast to traditional batch-based scheduling. Besides, GEAS aims to maximize the size of each formed batch for the efficiency concern. We present the batch-forming algorithm in Algorithm 2. Note that GEAS forms an individual batch for each consistency constraint and thus Algorithm 2 maintains multiple batches in parallel.

For each consistency constraint s, Algorithm 2 decides whether to initiate a new batch for the new context change chg_{new}. If the s-condition-matching function (i.e., Algorithm 1) returns True for chg_{new} with respect to constraint s (Line 2), the change is used to initiate a new batch $s.newBatch$ (Line 3), meaning that chg_{new} should not be checked with earlier collected changes. Otherwise, the function returns False, and chg_{new} is appended to s's current batch $s.batch$ (Line 5), meaning that this change can be safely checked together with earlier changes.

Algorithm 2 splits a sequence of context changes into isolated batches for each consistency constraint. We now show that such formed batches will not miss any context inconsistency, i.e., GEAS fulfills our preceding formulated quality objective (in Section 2.4). To show it, we have the following equivalence theorem:

Theorem 2 (**Equivalence of inconsistency detection result).** *Given any consistency constraint and context pool, GEAS always returns the same inconsistency detection result as immediate scheduling does for any sequence of context changes.* □

Proof. Given a sequence of context changes, GEAS splits it into multiple batches and checks each batch in turn. On the other hand, immediate scheduling checks each change in turn. The theorem essentially claims that GEAS always returns the same inconsistency detection result for each batch as those accumulated by immediate scheduling via checking each change in the batch. As the batches can be multiple, we transform the problem into an equivalent one: GEAS returns the same inconsistency detection result (i.e., R_{ID} value) as immediate scheduling does for one batch of context changes, as long as their initial R_{ID} values are equal before checking this batch.

Let the consistency constraint be s, context pool be P, and initial time point be t_0. Then the claim of immediate scheduling's and GEAS's initial R_{ID} values being equal can be represented as follows:

$$R_{ID-imd}(s, P, t_0) = R_{ID-geas}(s, P, t_0).$$

Assuming this claim holding for now, we proceed with our proof and will come back to this claim later.

Then we consider a batch of context changes formed by GEAS, chg_1, chg_2, ..., chg_m, which are collected at time points $t_1, t_2, \ldots, t_m$, respectively. We now prove that GEAS will return the same R_{ID} value as immediate scheduling does after checking these changes, i.e.,

$$R_{ID-imd}(s, P, t_m) = R_{ID-geas}(s, P, t_m).$$

Since constraint s and pool P do not change in our proof, for ease of presentation, we omit their presence in the R_{ID} and later R_{CC} representations. That is, the preceding two equations can be simplified as:

$$R_{ID-imd}(t_0) = R_{ID-geas}(t_0), \quad (1)$$
$$R_{ID-imd}(t_m) = R_{ID-geas}(t_m). \quad (2)$$

Note that the theorem also implies that the constraint checking used in immediate scheduling and GEAS should be the same or equivalent (i.e., equal R_{CC} values). We note that this is true, although we will explain more details later in Section 3.5. With the implication holding, we have the following equation (since R_{CC-imd} and $R_{CC-geas}$ values are always equal to each other, we refer to them by R_{CC} directly later):

$$R_{CC-imd}(t_i) = R_{CC-geas}(t_i).$$

To check the batch of context changes, chg_1, chg_2, ..., chg_m, if one uses immediate scheduling, its R_{ID} value would be:

$$R_{ID-imd}(t_m) = (\cup_{k=1}^{m} R_{CC}(t_k)) \cup R_{ID-imd}(t_0). \quad (3)$$

On the other hand, if one uses GEAS, its R_{ID} value would be:

$$R_{ID-geas}(t_m) = R_{CC}(t_m) \cup R_{ID-geas}(t_0). \quad (4)$$

The key difference between the preceding two equations lies in $\cup_{k=1}^{m} R_{CC}(s, P, t_k)$ and $R_{CC}(s, P, t_k)$. So we study their relations as below. We first claim the following relation holding ($1 \leq a \leq b < m$):

$$R_{CC}(t_a) \cup R_{CC}(t_b) \subseteq R_{CC}(t_{a-1}) \cup R_{CC}(t_{b+1}). \quad (5)$$

To see it, if this relation does not hold, there must exist one context inconsistency, e.g., inc_x, which belongs to $R_{CC}(t_a) \cup R_{CC}(t_b)$ but does not belong to $R_{CC}(t_{a-1}) \cup R_{CC}(t_{b+1})$. This indicates that inc_x is caused by some change in the batch between chg_a and chg_b (inclusive), and becomes undetectable by another change no later than chg_{b+1}. Let the change causing inc_x be chg_x and the one causing inc_x undetectable be chg_y. Then we have $a \leq x < y \leq b + 1$, and chg_x must be an inc+ or inc? change and chg_y be an inc− or $inc?$ change. According to the definition of s-condition, the pair of chg_x and chg_y becomes one of s's s-conditions. As a result, the two changes, chg_x and chg_y , must be isolated into two batches. Since this violates the fact (chg_x and chg_y are in the same batch), relation (5) must hold.

We use relation (5) recursively to construct the following new relation by replacing a and b with i and $m - i$, then $i - 1$ and $m - i + 1$, ..., until 1 and $m - 1$ ($1 \leq i \leq \lfloor m/2 \rfloor$):

$$\begin{aligned} & R_{CC}(t_i) \cup R_{CC}(t_{m-i}) \\ & \subseteq (R_{CC}(t_{i-1}) \cup R_{CC}(t_{m-i+1})) \\ & \subseteq (R_{CC}(t_{i-2}) \cup R_{CC}(t_{m-i+2})) \\ & \ldots \\ & \subseteq (R_{CC}(t_0) \cup R_{CC}(t_m)). \end{aligned} \quad (6)$$

We now use relation (6) to study earlier discussed key difference, $\cup_{k=1}^{m} R_{CC}(s, P, t_k)$ and $R_{CC}(s, P, t_k)$. It depends on whether m is odd or even.

Case (m is odd). Then the valid range for i is from 1 to $(m-1)/2$ (inclusive) for relation (6). By applying the relation $(m - 1)/2$ times, we obtain the following relation (pairing $R_{CC}(t_1)$ and $R_{CC}(t_{m-1})$, then $R_{CC}(t_2)$ and $R_{CC}(t_{m-2})$, ..., until $R_{CC}(t_{(m-1)/2})$ and $R_{CC}(t_{(m+1)/2})$:

$$\begin{aligned} & \cup_{k=1}^{m} R_{CC}(t_k) \\ & = R_{CC}(t_1) \cup \ldots \cup R_{CC}(t_{m-1}) \cup R_{CC}(t_m) \\ & \subseteq R_{CC}(t_0) \cup \ldots \cup R_{CC}(t_m) \cup R_{CC}(t_m) \\ & \ldots \\ & \subseteq R_{CC}(t_0) \cup R_{CC}(t_m). \end{aligned}$$

Case (m is even). The valid range for i is from 1 to $m/2$ (inclusive). The only difference from the preceding case is $R_{CC}(t_{m/2})$, which is left without pairing. We solve the

problem by cloning $R_{CC}(t_{m/2})$ once, so that relation (6) can be applied similarly:

$$\begin{aligned}
&\cup_{k=1}^{m} R_{CC}(t_k) \\
&= (\cup_{k=1}^{m/2-1} R_{CC}(t_k)) \cup (\cup_{k=m/2+1}^{m} R_{CC}(t_k)) \\
&\quad \cup R_{CC}(t_{m/2}) \cup R_{CC}(t_{m/2}) \\
&\ldots \\
&\subseteq R_{CC}(t_0) \cup R_{CC}(t_m) \cup R_{CC}(t_0) \cup R_{CC}(t_m) \\
&= R_{CC}(t_0) \cup R_{CC}(t_m).
\end{aligned}$$

Combining two cases, for any m, the following relation holds:

$$\cup_{k=1}^{m} R_{CC}(t_k) \subseteq R_{CC}(t_m) \cup R_{CC}(t_0).$$

We add equation (1) to the above relation by the set union operation, and obtain the following new relation:

$$\begin{aligned}
&(\cup_{k=1}^{m} R_{CC}(t_k)) \cup R_{ID-imd}(t_0) \\
&\subseteq R_{CC}(t_m) \cup R_{CC}(t_0) \cup R_{ID-geas}(t_0).
\end{aligned}$$

We further simplify it by the fact that $R_{CC}(t_0) \subseteq R_{ID-geas}(t_0)$, since the latter is the accumulation result including the former, as we defined earlier in Section 2.5:

$$\begin{aligned}
&(\cup_{k=1}^{m} R_{CC}(t_k)) \cup R_{ID-imd}(t_0) \\
&\subseteq R_{CC}(t_m) \cup R_{ID-geas}(t_0). \qquad (7)
\end{aligned}$$

On the other hand, it is easy to observe that $\cup_{k=1}^{m} R_{CC}(t_k)$ includes $R_{CC}(t_m)$ naturally. Then the following relation trivially holds:

$$\begin{aligned}
&(\cup_{k=1}^{m} R_{CC}(t_k)) \cup R_{ID-imd}(t_0) \\
&\supseteq R_{CC}(t_m) \cup R_{ID-geas}(t_0). \qquad (8)
\end{aligned}$$

By combining relations (7) and (8), we eventually obtain the following equation:

$$\begin{aligned}
&(\cup_{k=1}^{m} R_{CC}(t_k)) \cup R_{ID}(t_0) \\
&= R_{CC}(t_m) \cup R_{ID}(t_0). \qquad (9)
\end{aligned}$$

In equation (9), the left side is $R_{ID-imd}(s, P, t_m)$ and the right side is $R_{ID-geas}(s, P, t_m)$ (recall equations (3) and (4)). Therefore,

$$R_{ID-imd}(t_m) = R_{ID-geas}(t_m).$$

Thus, given any batch of context changes formed by GEAS, it always returns the same R_{ID} value as immediate scheduling does, if their initial R_{ID} values before checking this batch are equal (referred to as the *intermediate proof*). Now come back to our earlier claim that GEAS and the immediate scheduling have their initial R_{ID} values equal. We say that this claim always holds, because: (1) if the "initial" refers to the very beginning when no batch has ever been checked, then both initial R_{ID} values are trivially equal to $\varnothing$; (2) otherwise, the "initial" refers to a time point where several earlier batches have been checked, and with the preceding point (1) and the intermediate proof, one can easily obtain the "equal initial R_{ID} value" result.

Combining these pieces and back to our theorem, for any sequence of context changes, which can form one or multiple batches, GEAS always returns the same inconsistency detection result as immediate scheduling does. This completes the proof. □

Algorithm 3: (scheduling) Scheduling constraint checking

Input: set of consistency constraints $\mathcal{S}$, context changes in a stream $chgStream$
Output: inconsistency detection result R_{ID}

```
1  while isNotEmpty(chgStream) do
2      // form batches and check changes
3      chg_new ← getNext(chgStream)
4      batch-forming(S, chg_new)
5      for each s ∈ S do
6          if s.newBatch ≠ null then
7              R_ID ← checking(s, s.batch) ∪ R_ID
8              s.batch ← s.newBatch
9              s.newBatch ← null
10 for each s ∈ S do
11     // clean up
12     if s.batch ≠ null then
13         R_ID ← checking(s, s.batch) ∪ R_ID
14         s.batch ← null
15 return R_ID
```

As a summary, in this step GEAS forms batches by adaptive grouping of collected context changes. We also prove that such formed batches will not miss any context inconsistency, justifying its quality objective. In the following, we explain how to schedule constraint checking for these formed batches.

3.5 Step 4: Scheduling Constraint Checking

Given a formed batch of context changes, GEAS schedules constraint checking for all changes in this batch. Then constraint checking techniques (e.g., ECC [8], PCC [6], ConC [10], and GAIN [11]) should support checking multiple context changes as a whole. Note that not all existing constraint checking techniques support this naturally. In this step, we explain how to adapt these techniques, if necessary, for this purpose.

Algorithm. We first present the whole constraint checking scheduling process in Algorithm 3, in which the checking function can refer to any existing constraint checking technique (we will discuss it later). In the algorithm, GEAS examines each context change collected in the stream $chgStream$ (Line 3), and decides whether it initializes a new batch or is put into the existing batch for each consistency constraint by function batch-forming, i.e., Algorithm 2 (Line 4). Then, if any constraint has a new batch initialized (Line 6), which indicates that its existing batch is ready for checking, GEAS schedules a specific constraint checking technique to check all changes in the existing batch for this constraint and adds its detected context inconsistencies into the final result R_{ID} (Line 7). Finally, when the stream has no more context change left, GEAS cleans up the remaining changes in the existing batch for each constraint (Lines 12–14), and returns the final result (Line 15).

Now we focus on the checking function in the algorithm, which can refer to any constraint checking technique but requires it able to check multiple changes in a batch as

a whole for one consistency constraint. We examine existing constraint checking techniques (i.e., ECC [8], PCC [6], Con-C [10], and GAIN [11]) and partition them into two categories: *non-cache-based* and *cache-based*. A technique is *non-cache-based* if it does not cache any previous checking result, and *cache-based* if it has to cache previous results for speeding up the calculation of new checking results. ECC [8], Con-C [10], and GAIN [11] belong to the former, and PCC [6] belongs to the latter. For non-cache-based techniques, GEAS can apply to them almost directly, as there is no essential difference for these techniques to apply one or multiple changes to the context pool before checking the concerned contexts. For cache-based techniques, they may need some adjustment, when they work incrementally and the incremental granularity is based on a single change.

PCC adaptation. We now explain how to adapt PCC [6] to work with GEAS. PCC's incremental granularity is based on a single change, and we extend it to support checking multiple context changes as a whole. We name the extended PCC PCC_m. PCC_m needs to account for two key tasks, namely, *truth value evaluation* and *link generation*. The former evaluates the truth value of a consistency constraint with respect to the contexts in a pool, and the latter generates links to explain how a constraint has been satisfied or violated (Section 2.2). To enable PCC_m to support the two tasks with respect to multiple context changes as a whole, we slightly modify PCC_m's truth value evaluation and link generation semantics over those of PCC.

A constraint checking technique's truth value evaluation and link generation semantics formally specify how the truth value evaluation and link generation should be conducted for detecting context inconsistencies. We give one example (universal formula) in Fig. 3 for PCC. $\mathcal{T}$ is the truth value evaluation function that takes a formula (e.g., "$\forall v \in C(f)$") and a variable assignment (e.g., α) as input, and returns the truth value of this formula under this particular variable assignment. Similarly, $\mathcal{L}$ is the link generation function that takes the same input but returns a set of links to explain the truth value from the $\mathcal{T}$ function. PCC checks a context change for a universal formula by four cases, namely, fully reusable (the change does not affect C and subformula f at all), checking a single addition change (affecting C), checking a single deletion change (affecting C), and checking a change that affects subformula f instead. These four cases incur 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. For a detailed explanation and its examples, interested readers can refer to PCC's original publication [6].

We give the corresponding new truth value evaluation and link generation semantics in Fig. 4 for PCC_m. PCC_m checks multiple context changes as a whole. Similarly, it also partitions the checking into four cases. The first (fully reusable, i.e., no affection at all) and last (affecting subformula only) cases are exactly the same as before, but the second and third cases now need to take care of more situations. When multiple context changes are checked as a whole, the situation can be checking addition changes only, deletion changes only, or both addition and deletion changes. In PCC_m, the second case is for the first situation (i.e., checking addition changes only), and the third case is for the second and third situations (i.e., checking deletion changes only, or both addition and deletion changes), which are merged for simplifying the semantics (i.e., as the case where there is at least one deletion change).

Similarly, the semantics for an existential formula can also thus be modified to support checking multiple context changes as a whole, but those for other formulas keep the same as in PCC, since those formulas will not be directly affected by any context change. As a whole, one can easily observe that such modifications are essentially immaterial to the truth value evaluation and link generation semantics of PCC and PCC_m. Therefore, PCC_m will return the same checking results as PCC.

Efficiency analysis. Regarding the efficiency, PCC_m will take almost the same time to check all involved context changes as PCC (in this case PCC will be scheduled multiple times), since each change still needs to be checked. Nevertheless, PCC_m can still win over PCC as accumulating the checking results for multiple changes can now be done in one run instead of multiple runs. Regarding ECC, Con-C, and GAIN, now they work much faster when combined with GEAS. This is because they can spend the same amount of time on checking multiple context changes instead of one change, since they work in a non-cache-based way. Thus their efficiency improvement relates to the reduced scheduling of constraint checking, which is close to $(n - m) \,/\, m$, where n is the number of changes and m is the average number of scheduled constraint checking. Note that this is the rough estimation of the efficiency improvement (assuming that each checking takes the same time). This efficiency analysis will be validated by our later evaluation.

As a summary, in this step GEAS copes with existing constraint checking techniques for efficient context inconsistency detection with quality guarantee. In the following, we consider optimizing GEAS by identifying and removing impact-opposite context changes in GEAS-maintained batches.

4 GEAS Optimization

In this section, we optimize GEAS for further efficiency improvement by a dedicated change-cancellation technique. The optimization applies to scenarios where there are only context addition and deletion changes. We note that such scenarios are quite common, and in fact update changes can also be decomposed into deletion and addition changes.

4.1 Optimization Overview

GEAS realizes efficient context inconsistency detection by forming adaptive batches that never contain context changes of latent interference with each other. In this section, we optimize GEAS by identifying and removing change pairs from the batches as long as they contain opposite impacts in constraint checking. In this case, we say that they form an *inter-cancellation relationship* and we model such relationship as a *cancellation condition* (*c-condition*). By c-conditions, one can greatly reduce the number of context changes that have to be checked, thus further improving the efficiency of context inconsistency detection.

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

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) $\mathcal{T}_0[\forall v \in C(f)]_\alpha \wedge \mathcal{T}[f]_{\mathsf{bind}((v,x),\alpha)} \mid \{x\} \in C - C_0$,
   if C has an addition changes;
3) $\top \wedge \mathcal{T}_0[f]_{\mathsf{bind}((v,x_1),\alpha)} \wedge \ldots \wedge \mathcal{T}_0[f]_{\mathsf{bind}((v,x_n),\alpha)} \mid x_i \in C$,
   if C has a deletion change;
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) $\mathcal{L}_0[\forall v \in C(f)]_\alpha \cup \{l | l \in \{(\mathsf{violated}, \{(v,x)\})\} \otimes \mathcal{L}[f]_{\mathsf{bind}((v,x),\alpha)} \} \mid \{x\} = C - C_0 \wedge \mathcal{T}[f]_{\mathsf{bind}((v,x_i),\alpha)} = \bot$,
   if C has an addition changes;
3) $\{l | l \in \{(\mathsf{violated}, \{(v,x_i)\})\} \otimes \mathcal{L}_0[f]_{\mathsf{bind}((v,x_i),\alpha)} \} \mid x_i \in C \wedge \mathcal{T}[f]_{\mathsf{bind}((v,x_i),\alpha)} = \bot$,
   if C has a deletion change;
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 semantics for a universal formula in PCC

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

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. 4: Truth value evaluation and link generation semantics for a universal formula in PCC_m (blue parts are newly added or modified)

We give an overview of our GEAS optimization in Fig. 5. Note that we replace the original Step 3 in Fig. 2 with the parts (Steps 3, 5, and 6) in the dashed rectangle in Fig. 5. Step 5 (Section 4.2) and Step 6 (Section 4.3) are new. The former infers c-conditions, and the latter examines context changes against these inferred c-conditions. Then, Step 3 (Section 4.4) is updated to form and refine batches by adaptively grouping collected context changes according to matching results of s-conditions and examining results of c-conditions. We in the following introduce these steps in turn.

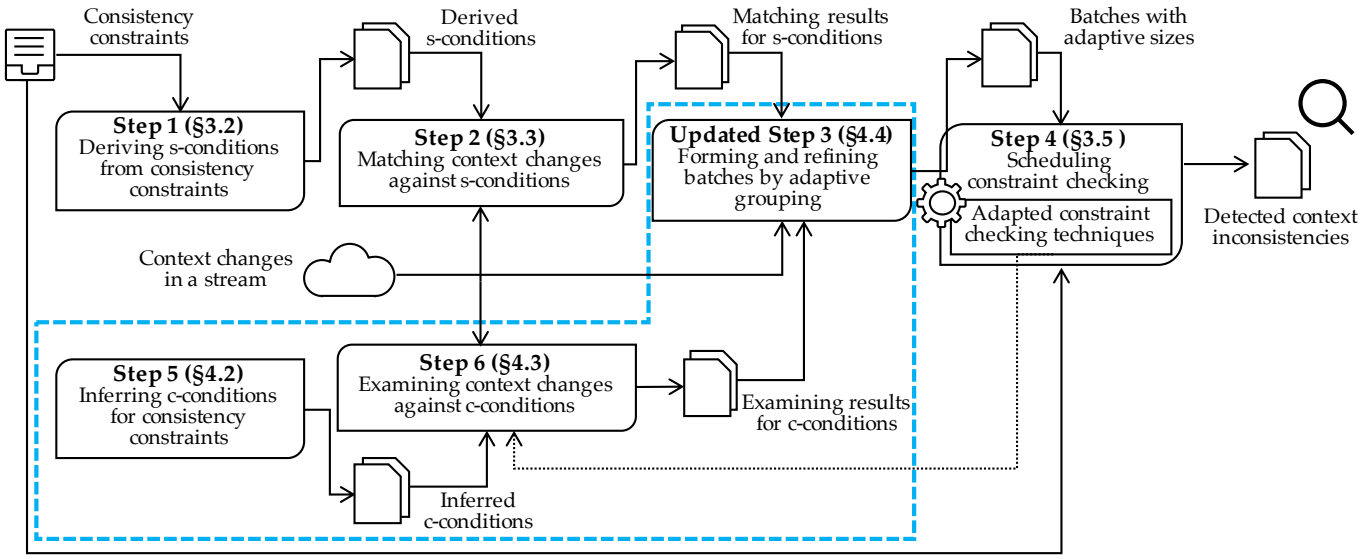


Fig. 5: GEAS optimization overview.

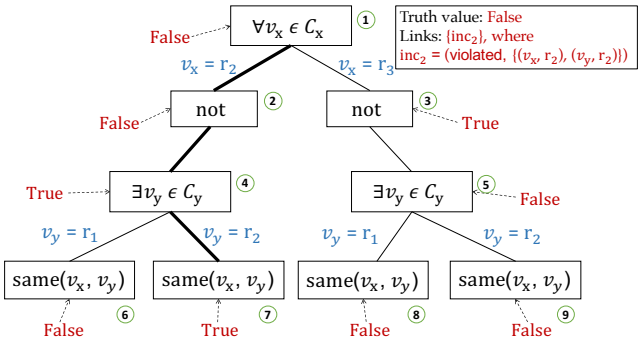


Fig. 6: A runtime tree example for constraint S_{loc}.

4.2 Step 5: Inferring Cancellation Conditions for Consistency Constraints

Original GEAS (later referred to as GEAS-ori) forms batches of context changes for constraint checking. Changes in such batches are already safe for checking together, not missing any context inconsistency. Nevertheless, optimized GEAS (later referred to as GEAS-opt) further removes change pairs from the batches if such pairs cause opposite impacts in constraint checking, thus reducing the changes that have to be checked. By "opposite impact", we mean that one change in a pair can incur some intermediate results (in terms of partial truth values and links), while the other change in this pair exactly destroys these intermediate results.

For better understanding, we use the notion of *runtime tree* (supported in existing constraint checking techniques [6], [10], [11]) to explain such opposite impact. We note that constructing and maintaining such tree-alike structures are all done by these checking techniques internally, and that GEAS only observes their internal states and makes minor updates as we explain later. Still, we briefly explain how a runtime tree is constructed with respect to a given consistency constraint and elements in its associated contexts [10]. In a runtime tree, a universal or existential formula would take multiple branches, whose number is equal to that of elements in this formula's associated context. Each branch is labeled with a particular element from the concerned context and assigned to this formula's associated variable. These branches represent this formula's subformula with different variable assignments. An "and", "or", and "implies" formula would take two branches, each of which corresponds to one of its subformulas. A "not" formula takes one branch, which corresponds to its only subformula. Terminal $bfunc$ is always a leaf node because it does not have any subformula.

Now consider our earlier consistency constraint S_{loc}: $\forall v_x \in C_x$ (not ($\exists v_y \in C_y$ (same(v_x, v_y)))). Fig. 6 shows its corresponding runtime tree (one example). The tree shows how the constraint is checked with respect to the contexts in a pool. For this example, $C_x = \{r_2, r_3\}$ and $C_y = \{r_1, r_2\}$, which corresponds to the scenarios of P_8 in Fig. 1. We also annotate intermediate truth values and links for all nodes in Fig. 6. Now suppose that the next two context changes are $<-, C_x, r_3>$ and $<+, C_x, r_4>$. One can observe that applying and checking the first change $<-, C_x, r_3>$ would cause deleting the whole right branch including nodes 3, 5, 8, and 9, while applying and checking the second change $<+, C_x, r_4>$ would add this branch completely back, except variable assignment $v_x = r_3$ being changed to $v_x = r_4$, without affecting the final checking results (i.e., False and $\{inc_2\}$). In this case, we say that the two changes cause opposite impacts in constraint checking. If one removes the two changes from the batch, as if they were not present, the efficiency of context inconsistency detection can be improved without any negative consequence. This is our planned change-cancellation technique for GEAS optimization.

The key to realize change cancellation is to identify those change pairs in a batch, whose impacts on constraint checking are exactly opposite to each other. The impact covers both evaluated truth values and generated links. However, a complete validation on whether two changes have exactly opposite impacts on constraint checking requires evaluating truth values and generating links for related nodes on runtime trees, which would spend valuable time, weakening

our targeted optimization. Considering that link generation is much more complex than truth value evaluation [6], [37], our idea is to: (1) only evaluate and compare truth values for the nodes affected by given changes, and (2) protect *critical nodes* whose truth values relate to the final checking results (e.g., links) from being affected by given changes.

Step 5.1: Evaluating and comparing truth values. Given a pair of context changes, this sub-step examines whether the nodes related to them on runtime trees would obtain exactly the same truth values. For example, the aforementioned change pair, $<-$, C_x, $r_3>$ and $<+$, C_x, $r_4>$, both relate to the same four nodes (3, 5, 8, and 9) in Fig. 6. One needs to evaluate truth values for these four nodes and compare whether their counterparts have equal values. We note that: (1) one can only evaluate and compare truth values for *bfunc* nodes, as they are the only places that can cause different values; (2) the pair of changes should concern the same context (e.g., $<-$, C_x, $r_3>$ and $<+$, C_x, $r_4>$ both concern context C_x), otherwise they cannot be compared. This can speed up our comparison.

If the pair of context changes fails the comparison, they are not impact-opposite. Otherwise, they pass the comparison, GEAS-opt would further rely on Step 5.2 to decide whether the change pair can be canceled. We also note that this sub-step invokes "adapted constraint checking techniques" in Step 4, as the dashed arrow shows in Fig. 5.

Step 5.2: Protecting critical nodes. The preceding sub-step validates only truth values for nodes affected by a given pair of context changes. We still need to ensure that links in the final checking results will not be affected by canceling the pair of changes. Note that since Step 5.1 guarantees that all affected nodes obtain the equal truth values for the change pair, then it is impossible to encounter cases where new inconsistencies are detected (e.g., $\{inc_a\} \rightarrow \{inc_a, inc_b\}$) or existing inconsistencies are gone (e.g., $\{inc_a, inc_b\} \rightarrow \{inc_a\}$) after applying the change pair (otherwise there must exist one node whose truth value is different for the change pair). The only remaining case is that detected inconsistencies might be updated (e.g., $\{inc_a\} \rightarrow \{inc_b\}$) after applying the pair (with all truth values not changed after applying the change pair). We need to invalidate this possibility to ensure that links are also not affected by the change pair.

As mentioned, for efficiency consideration, we do not generate and compare links. We use a light-weight technique by the notion of critical node to invalidate this possibility. We identify and protect those nodes from being affected by the given change pair, named *critical nodes*, as long as their truth values can determine the final checking results on runtime trees. Consider the earlier example in Fig. 6, where nodes 1, 2, 4, and 7 are critical nodes (we will discuss how to decide critical nodes soon later). If a given change is $<-$, C_x, $r_3>$, it would cause deleting the branch of $v_x = r_3$, but does not affect the truth value of any critical node (still False, False, True, and True for nodes 1, 2, 4, and 7, respectively). Then the change will not cause the aforementioned possibility, and thus can safely be considered in canceled change pairs. However, if a given change is $<-$, C_y, $r_2>$, it would cause deleting two branches of $v_y = r_2$, and will affect the truth values of all the four critical nodes (now True, True, and False for nodes 1, 2, and 4, with node 7 gone). Then the change can cause the aforementioned possibility, and thus should not be considered in canceled change pairs.

The remaining two issues are: (1) how to decide which nodes are critical nodes on a runtime tree, and (2) how to decide whether any critical node will be affected by a context change. For (1), since the last checking result (i.e., truth value and links, or R_{CC}) are available before checking a batch of context changes, one can easily identify the paths that generate the links in R_{CC} and consider all nodes on the paths as critical nodes. For the example in Fig. 6, inc_2 corresponds to the paths as annotated by bold line segments, from nodes 1, 2, 4, to 7, and all the four nodes are critical nodes, as mentioned earlier. For (2), since we aim to invalidate the possibility of $\{inc_a\} \rightarrow \{inc_b\}$, we choose to prevent those changes that can cause $\{inc_a\}$ gone to be included into canceled change pairs. Then we only examine whether any critical node will be deleted due to a change (truth values of other critical nodes will be affected accordingly). For the example in Fig. 6, change $<-$, C_y, $r_2>$ will cause node 7 to be deleted, and thus the change will affect this critical node as well as truth values of its upper-layer nodes, as mentioned earlier.

Combining the preceding efforts together, we infer c-conditions for deciding change pairs that can be canceled in a batch for each consistency constraint. A constraint's c-condition consists of three parts, as follows:

Definition 5 (Cancellation condition or c-condition). *Given a consistency constraint, its c-condition consists of the following three parts: considering a pair of addition-deletion or deletion-addition context changes, (1) the two changes are consecutive and appear at the head of the constraint's batch, (2) truth values are all equal for any pair of corresponding nodes on the constraint's runtime tree related to the two changes, and (3) no critical node on the runtime tree will be affected by either change.*

The (2) and (3) parts correspond to the aforementioned Step 5.1 and 5.2 examinations, which decide that the change pair will cause opposite impacts on constraint checking. The (1) part is the precondition, but can be much more relaxed (we will discuss it later in 4.3). When a pair of context changes satisfy all the three parts, we say that they form an *inter-cancellation relationship* and can be safely removed as if they were not present in the batch.

Let us revisit the example in Fig. 6, which concerns consistency constraint S_{loc} and its next two context changes $<-$, C_x, $r_3>$ and $<+$, C_x, $r_4>$. First, the two changes naturally satisfy the (1) part in the definition. Second, as analyzed earlier, replacing r_3 in C_x with r_4 does not change the truth value for any node on the right branch, and thus the (2) part is satisfied. Third, the second change $<+$, C_x, $r_4>$ does not affect any critical node on the left branch, as we analyzed for the first change $<-$, C_x, $r_3>$ in Step 5.2, and thus the (3) part is also satisfied. Therefore, this change pair satisfies constraint S_{loc}'s c-condition and is qualified for removal from S_{loc}'s batch.

As a summary, in this step GEAS-opt infers c-conditions for consistency constraints. The c-conditions are used for examining context changes at runtime. We have explained restricted cases of examining a pair of heading, consecutive context changes against a constraint's c-condition. In the fol-

lowing, we discuss how to relax the "heading, consecutive" precondition, so that the optimization can apply to more context change pairs.

4.3 Step 6: Examining Context Changes against Cancellation Conditions

The c-condition optimization aims to refine context changes in a batch by removing those impact-opposite change pairs. However, its "heading, consecutive" precondition makes its applicability very restricted: unless the batch contains only one change chg_x currently, one has no chance to examine its next change chg_y (since only in this case, (chg_x, chg_y) forms a heading, consecutive change pair).

To address this restriction, we plan to remove the "heading, consecutive" precondition. This implies that one would be allowed to remove any pair (chg_x, chg_y) formed from all changes in one batch (chg_1, chg_2, ..., chg_n) and its next change chg_{n+1}, as long as the pair satisfies the (2) and (3) parts of a c-condition. This further implies that the two changes chg_x and chg_y for removal would be as if they were heading, consecutive in the batch. Then it would require that checking the original batch (chg_1, chg_2, ..., chg_{n+1}) would return the same constraint checking result as checking the new batch, say, (chg_2, chg_{n+1}, chg_1, chg_3, ..., chg_{n-1}, chg_n), assuming that $x = 2$, $y = n + 1$. For this purpose, we have the following "order equivalence" theorem to guarantee this moving.

Theorem 3 (**Order equivalence).** *Given a consistency constraint s and its current batch of context changes, if any change pair from the batch satisfies the (2) and (3) parts of s's c-condition, then no matter whether one moves the change pair to the head of the batch or not, checking the whole batch of changes against s always returns the same constraint checking result.* □

Proof. As mentioned earlier, checking one batch of context changes against a consistency constraint takes two steps: (1) applying all changes in the batch to contexts in the pool, and (2) checking the updated contexts against the constraint for any context inconsistency. We in the following prove that moving the change pair does not affect both steps.

When the (1) step goes smoothly, all contexts in the pool are updated properly. Then the (2) step is straightforward for the order equivalence since it does not depend on the order at all. Its only assumption is that the (1) step's resulting contexts are the same no matter whether one moves the change pair or not. We in the following prove that this assumption holds.

To prove the assumption, we need to show that: (1) the new batch of context changes resulting from moving the change pair is still *valid* with respect to element operations in the changes to concerned contexts, and (2) the contexts resulting from the new batch are the same as those resulting from the original batch. When (1) holds, (2) is trivial since all changes in the new and original batches are the same and can be validly applied. We in the following focus on (1).

We first explain the validity of a batch of context changes with respect to element operations in the changes to concerned contexts. Consider a consistency constraint s and its current batch of context changes (chg_1, chg_2, ..., chg_n). We say that the batch is *valid* if neither of the following two cases exists: (1) deleting an element that does not exist from a context; (2) adding an element that already exists into a context. We then need to prove that moving the change pair that satisfies the (2) and (3) parts of constraint s's c-condition will not breach the validity of s's batch, i.e., if s's original batch is valid, then the new batch after moving the change pair is still valid. However, this target is more than necessary for the order equivalence. In fact, one only needs to prove that after moving the change pair to the head of the batch, the preceding two invalidity cases either do not exist or will not affect the equivalence of contexts resulting from either the new batch or the original batch. We prove it below.

The two changes in the pair must be a deletion change and an addition change, respectively. We consider the impact of moving them to the head of the batch in turn.

Moving the deletion change. Let the change be $<-$, C, $a>$. It is easy to observe that moving the deletion change concerns only the first invalidity case (i.e., deleting an element that does not exist from a context). This case contains two possibilities: (a) element a does not exist in context C at the head of the batch, and (b) after the deletion change is moved to the head of the batch, there exists another later deletion change on element a as well, without any prior change of adding a back to C. For (a), this is impossible because element a must already exist before applying this batch of context changes, since otherwise this deletion change cannot be qualified for cancellation (the (2) part of the c-condition guarantees it). For (b), this is also impossible, as since the original batch of context changes is valid, it must contain a change sequence of $<-$, C, $a>$, $<+$, C, $a>$, and $<-$, C, $a>$ (may not be consecutive; the second $<-$, C, $a>$ is the deletion change to move), and this sequence will certainly cause an impact sequence of inc$-$, inc$+$, and inc$-$, or inc$+$, inc$-$, and inc$+$, either of which already violates GEAS's s-conditions that form this batch. Therefore, the preceding invalidity cases do not exist if one moves the deletion change to the head of the batch.

Moving the addition change. Let the change be $<+$, C, $a>$. It is easy to observe that moving the addition change concerns only the second invalidity case (i.e., adding an element that already exists into a context). This case also contains two possibilities: (a) element a already exists in context C at the head of the batch, and (b) after the addition change is moved to the head of the batch, there exists another later addition change on element a as well, without any prior change of deleting a from C. For (a), since the original batch of context changes is valid, it must contain $<-$, C, $a>$ and then $<+$, C, $a>$ (may not be consecutive; $<+$, C, $a>$ is the addition change to move). So now the situation is: for the original batch, we have a existing in C, followed by changes $<-$, C, $a>$ and then $<+$, C, $a>$; for the new batch, we also have a existing in C, but followed by changes $<+$, C, $a>$ and then $<-$, C, $a>$. It seems invalid for the new batch, but if one allows a duplicated a temporarily existing in C, the invalidity will soon disappear automatically when applying change $<-$, C, $a>$. So our treatment is to support such temporary duplication, which has no negative consequence to constraint checking later. For (b), this is impossible, as since the original batch of context change is valid, it must contain a change sequence of $<+$, C, $a>$, $<-$, C, $a>$, and $<+$, C, $a>$ (may not be

consecutive; the second $<+, C, a>$ is the addition change to move), and this sequence will certainly cause an impact sequence of inc+, inc−, and inc+, or inc−, inc+, and inc−, either of which already violates GEAS's s-conditions that form this batch. Therefore, moving the addition change will neither incur any invalidity case nor affect the equivalence of contexts resulting from the new batch or the original batch.

As a summary, we prove that for the (1) step (applying changes), moving the change pair that satisfies the (2) and (3) parts of constraint s's c-condition to the head of the batch will neither incur any invalidity case nor affect the equivalence of contexts resulting from the new batch or the original batch. As such, the (2) step (checking contexts) will always return the same constraint checking results since the checked contexts are the same no matter whether one moves the change pair or not. This completes the proof for the order equivalence. □

Based on this theorem, one can safely remove the "heading, consecutive" precondition (i.e., the (1) part in the c-condition definition), so that GEAS-opt can apply to more context change pairs. In fact, our later evaluation shows that GEAS-opt can thus cancel around 80% changes from the batches formed by GEAS-ori. This enables a further improvement on the efficiency of context inconsistency detection, although there is some other overhead incurred by the optimization.

To use this change-cancellation technique, we present Algorithm 4 to explain the c-condition examination process. Note that for ease of presentation, we still name the relaxed c-condition (i.e., after removing the (1) part) as c-condition. The algorithm explains how to examine change pairs against c-conditions at runtime to identify those that can be removed without affecting inconsistency detection results. For the efficiency concern, the examination process works in an incremental way, similar to our aforementioned s-condition matching process in Section 3.3.

In the algorithm, each pair formed by context changes from consistency constraint s's batch and the new change chg_{new} is examined to see whether it satisfies the (2) and (3) parts of s's c-condition. For part (2), we use the evaluating function to validate whether the two changes in the pair always produce equal truth values for all concerned $bfunc$ nodes on constraint s's runtime tree based on its last status $lastTree$ (Lines 4–6). Note that evaluating invokes "adapted constraint checking techniques" in Step 4 (Section 4.1), but concerns truth value evaluation only (i.e., no link generation), which is extremely efficient (Section 4.2). For part (3), we use the affecting function to ensure that no critical node on constraint s's runtime tree will be affected by the two changes in the pair (Line 8). One can observe that the (1) part is no longer involved, and that when the (2) and (3) parts are satisfied, the examining process will terminate with the concerned change chg_i from constraint s's batch to form the change pair for cancellation with the new change chg_{new} (Line 9). Otherwise, no change can be canceled and the process returns null.

As a summary, in this step GEAS-opt examines and identifies those context change pairs that satisfy our relaxed c-conditions for cancellation (with the support of our order equivalence theorem). Then the batches of context changes formed by GEAS-ori can be refined, and we explain the batch-refining process below.

Algorithm 4: (c-condition-examining) Examining context changes against c-conditions

Input: consistency constraint s, new context change chg_{new}
Output: examining result chg // the change to pair

```
1  chg ← null
2  for each chg_i ∈ s.batch do
3      // examine part (2)
4      tvs_1 ← evaluating(s.lastTree, chg_i)
5      tvs_2 ← evaluating(s.lastTree, chg_new)
6      if allEqual(tvs_1, tvs_2) then
7          // examine part (3)
8          if !affecting(s.lastTree, chg_i) &&
            !affecting(s.lastTree, chg_new) then
9              chg ← chg_i
10             break
11 return chg
```

Algorithm 5: (batch-forming-and-refining) Forming and refining batches by adaptive grouping and change cancellation

Input: set of consistency constraints $\mathcal{S}$, new context change chg_{new}
Output: set of consistency constraints $\mathcal{S}$ (updated)

```
1  for each s ∈ S do
2      if s-condition-matching(s, chg_new) then
3          s.newBatch ← <chg_new>
4      else
5          chg ←c-condition-examining(s, chg_new)
6          if chg == null then
7              s.batch ← append(s.batch, chg_new)
8          else
9              s.batch ← remove(s.batch, chg)
10             updating(s.lastTree, chg, chg_new)
11 return S
```

4.4 Updated Step 3: Forming and Refining Batches

Now based on the matching results of s-conditions and examining results of c-conditions, GEAS-opt can form and refine batches of context changes for constraint checking.

Note that this forming and refining process extends our aforementioned forming process (Algorithm 2 in Section 3.4). So it is an updated Step 3 (Section 4.1), and we present it in Algorithm 5. For each consistency constraint s, the algorithm decides whether to initiate a new batch for the new context change chg_{new} or allow this change into constraint s's existing batch, according to the matching results of s's s-conditions (Line 2). This part resembles its corresponding one in Algorithm 2. Nevertheless, for the later case, the process additionally examines whether there exists any pair formed by changes from constraint s's batch

and the new change chg_{new} that satisfies s's c-condition (Line 5). If no (null returned), chg_{new} will be appended to constraint s's existing batch (Line 7) as in Algorithm 2. Otherwise, s's c-condition is satisfied, and chg_{new} will be canceled (by no longer appending it to the batch) instead, together with its paired change (by the remove function) identified by c-condition-examining function (Line 9). Finally, the two elements (chg and chg_{new}) concerned in the two canceled changes need some follow-up actions by the updating function (Line 10), as we explained below.

For a pair of context changes for cancellation (chg and chg_{new}), as we analyzed earlier, although they may concern different elements (e.g., adding element a into context C and then deleting b from C), their impacts are opposite. For rare cases, the two elements (say, a and b) happen to be the same (i.e., $a = b$). Then there is nothing one has to follow after canceling them. However, for most cases, they are not the same. Then one has to eliminate any possible side effect (i.e., making the effect of canceling both changes exactly equal to not canceling them at all). Note that our optimization is based on the evaluation on all changes in a constraint's current batch, but does not account for any side effect caused by future changes not coming yet. For the preceding example (i.e., changes $<-$, C_x, $r_3>$ and $<+$, C_x, $r_4>$), canceling both changes will not affect constraint checking of all changes in the concerned constraint's batch, as we analyzed earlier. However, canceling them will lead to element r_3 remaining in the context pool, but not canceling them will lead to r_4 in the context pool. It is possible that they have different constraint checking results with future changes. Therefore, to eliminate such possible side effect (although not coming yet), one has to replace all occurrences of r_3 with those of r_4, so that everything will look as if no cancellation had occurred.

For non-cache-based constraint checking techniques (e.g., ECC, Con-C, and GAIN), one only needs to make the element replacement (e.g., from r_3 to r_4) in the context pool. For cache-based constraint checking techniques (e.g., PCC), such techniques conduct constraint checking incrementally based on both the context pool and runtime trees. Then, besides updating the context pool, one also needs to update related variable assignments (e.g., from $v_x = r_3$ to $v_x = r_4$) in all related runtime trees. The updating function in Algorithm 5 (Line 10) exactly conducts such replacements. We note that making such replacements is efficient, since: (1) updating the context pool is straightforward, (2) updating variable assignments concerns part of branch information only, not affecting any structure of the runtime trees, and (3) the most complex data structure concerns links maintained in cache-based constraint checking [6], but no link will involve any element in canceled change pairs (explained in Step 5.2 in Section 4.2), and thus links are free of any updating action.

As a summary, in this step GEAS-opt forms and refines batches for context changes based on both s-conditions and c-conditions. By additionally considering c-conditions, GEAS-opt cancels those change pairs that are unnecessary for constraint checking, without missing any inconsistency detection result with theoretical guarantee. Besides, with such cancellation, GEAS-opt reduces the number of context changes for checking, leading to further improved inconsistency detection efficiency. In the following, we experimentally evaluate our GEAS-ori and GEAS-opt.

5 Evaluation

In this section, we evaluate our GEAS's performance on a taxi application with large-volume real-world data.

5.1 Research Questions

In this work, we propose GEAS to improve context inconsistency detection. From our earlier problem formulation (Section 2.5), it contains two objectives: quality and efficiency. As such, GEAS's performance should be evaluated on whether and how the two objectives are fulfilled. To do so, we evaluate whether GEAS can fulfill highly-efficient zero-missing context inconsistency detection, i.e., whether GEAS can help existing constraint checking techniques to greatly improve the inconsistency detection efficiency without missing any inconsistency detection result. This aims for GEAS's effectiveness (RQ2). Besides, we are interested in GEAS's overhead, i.e., what time cost is required for GEAS's effectiveness in its offline analyses and online matching/examination, and whether the cost will compromise its effectiveness (RQ3). On the other hand, GEAS should also compare to existing strategies (i.e., immediate scheduling and batch-based scheduling) to justify its motivation and validate its unique advantages (RQ1 and RQ3). Therefore, we study the following three research questions in the evaluation.

We raised research questions as follows.

RQ1 (Motivation): *How serious is the inconsistency missing problem with the traditional batch-based scheduling in context inconsistency detection, as compared to immediate scheduling?*

RQ2 (Effectiveness): *How effective is GEAS in improving the efficiency of existing constraint checking techniques and in protecting inconsistency detection results, as compared to immediate scheduling and batch-based scheduling?*

RQ3 (Overhead): *How much time does GEAS take in its offline s-condition/c-condition analyses and online s-condition matching/c-condition examination?*

5.2 Experimental Design and Setup

To answer the three research questions, we first introduce the resources used in the experiments, which include the *subject* (a taxi application), *data* (taxi data used by the application), and *consistency constraints* (constraints for governing the consistency of the taxi data). We then explain the experimental *process* and *setup* with these resources. Finally, we explain *how to answer* the research questions through the designed process.

Subject. We selected a taxi application, SmartCity, with its large-volume real-world taxi data as our experimental subject. It was launched by an Urban Transport Planning Center for its smart city perspective in a city of South China. We selected this application because it has been used in previous research [6], [7], [10], [11], [12], and this facilitates our comparison.

The SmartCity application focuses on a city's traffic conditions for several hot areas as well as the whole city. For each hot area, the application maintains a collection

of recent taxi conditions (e.g., GPS data, taxi ID, driving speed, driving direction, and service status) for taxis driving inside this area. Such taxi conditions are regarded as the *context* associated with the hot area, which is subject to change from time to time. Based on all such contexts, the application evaluates the traffic conditions for hot areas as well as the whole city. Then it can provides smart services, like smart routing (i.e., recommending an optimal route to a destination for a requesting driver), and traffic jam avoidance (i.e., adjusting the route when there is any jam occurring ahead).

Data. For the SmartCity application, we obtained a total of 1.55 million taxi data (i.e., the aforementioned taxi conditions), which cover 760 distinct taxis monitored within a continuous period of 24 hours. These 1.55 million raw data correspond to 6.75 million *context changes* according to the context design in the application. Fig. 7 shows the distribution of context changes for 24 hour-based groups (0–23).

Each group i ($0 \leq i \leq 23$) includes all context changes from the start of an hour (inclusive) to that of its next hour (exclusive), starting from 11am. For example, group 0 represents the context changes collected in the time slot of 11am–12noon, and group 1 is for 12noon–1pm. We observe that the numbers vary greatly, from 180,218 to 365,688 (up to a 185,470 or 103% difference), and this makes them incur different workloads to constraint checking (e.g., lightest workload around 4am–6am, and the heaviest workload for rush hours 5pm–7pm). Besides, the taxi data also have different features in different groups, and this also affects the checking workload (e.g., after the midnight, most taxi conditions contain a "no service" tag, which makes some constraints evaluated to a different value, leading to a lighter checking workload). These features make the whole data set representative for evaluating different constraint checking techniques on their abilities against various workloads.

Consistency constraints. The SmartCity application was with 22 consistency constraints, from previous research [6], [7], [10], [11], [12], and originally from its developer, the Urban Transport Planning Center [6]. The constraints cover all seven formula types, which are complete to the constraint language (Section 2.2). The constraints are stored in the form of XML files for fetching and checking by our GEAS implementation.

These consistency constraints specify different necessary properties that must hold about the aforementioned contexts (e.g., taxi conditions about hot areas). Some constraints concern all monitored taxis in the city, and enforce speed limits on these taxis (e.g., no more than 200 km/h, which is a reasonable upper limit for the city [6]), or location restrictions (e.g., a taxi should not appear in a restricted area like sea). Other constraints concern only those taxis inside hot areas, and enforce their spatial restrictions (e.g., a taxi cannot appear in two distinct areas at the same time). Besides, these constraints may also check different fields in taxi conditions in the contexts (e.g., some constraints focus only on those taxis with a tag set to "on service" or "no service"). This makes these constraints incur different workloads in terms of checking complexity, and become suitable artifacts for evaluating different constraint checking techniques.

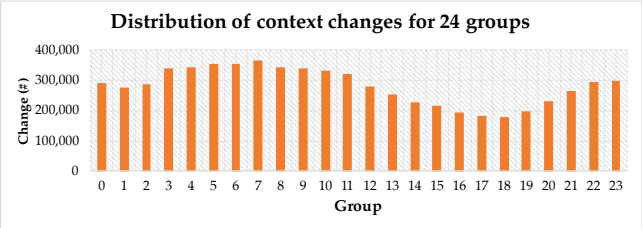


Fig. 7: Distribution of context changes for 24 hour-based groups

Note that the SmartCity application has a clear connection with the package delivery application discussed in our motivating example (Section 2.4), which contains several mapping relations: (1) regarding contexts, the locations of taxis in hot areas in the city can correspond to those of robots in warehouses; (2) regarding context changes, the location changes of taxis across different hot areas can correspond to those of robots across different warehouses; (3) regarding consistency constraints, the restrictions on location changes of taxis can correspond to those on locations changes of robots, and additionally the SmartCity application considers more properties about taxi attributes like speed and service status. These mapping relations help immigrate conceptually from the package delivery application in the motivating example to the SmartCity application in our evaluation.

Process. Our experiments contain two phases, namely, *preparation* and *execution*.

For the preparation phase, we derive s-conditions and c-conditions from the 22 consistency constraints for use by GEAS. For s-conditions, we derived a total of 340 s-conditions from these constraints (min: 4, max: 16, avg: 15.5). We observe that the derivation is extremely efficient (several milliseconds). For c-conditions, we have already inferred them in Section 4.2, and thus no extra time is required here.

For the execution phase, we implemented all scheduling strategies and constraint checking techniques in Java to facilitate experimental comparisons. In particular, all scheduling strategies are straightforward in their semantics and thus implemented in a single-threaded manner. Regarding constraint checking techniques, their implementations follow their built-in algorithms, e.g., single-threaded and non-incremental for ECC, single-threaded and incremental for PCC, multi-threaded with multi-CPU support for ConC, and multi-threaded with multi-GPU support for GAIN. We combined these implemented scheduling strategies and constraint checking techniques as a consistency management [38] service. In experiments, we used a middleware-based architecture following conventional consistency management [6]. That is, the SmartCity application runs on top of the middleware, and the middleware is responsible for feeding context changes to the application. The consistency management is integrated as part of services in the middleware and can be configured with any specific scheduling strategy and constraint checking technique working with given consistency constraints. We collected each combination's inconsistency detection results and spent time for comparisons from the middleware. We measure the actual time required by each combination for checking given context changes. To do so, we made all context changes fed to

the combination in a sequence and this process proceeded only when the last checking completed.

Setup. To follow the aforementioned experimental process (i.e., checking taxi context changes against consistency constraints), we design the following four *independent variables* related to the setting of each combination and its checking workload.

- *Scheduling strategy.* We study the three scheduling strategies discussed in this article, namely, immediate scheduling (IMD), batch-based scheduling (BAT), and our GEAS (two versions, i.e., GEAS-ori and GEAS-opt).
- *Batch size.* This is a secondary setting only for the BAT strategy, as BAT can be customized with different batch sizes as we discussed earlier. We controlled the batch size from 2 to 10, with a pace of 1. The minimal value 2 was selected because 1 would make BAT become IMD. The maximal value 10 was selected because it made BAT reach an inconsistency missing rate near 80%, which is already unacceptable.
- *Constraint checking technique.* We study the four constraint checking techniques, namely, ECC [8], Con-C [10], GAIN [11], and PCC [6] (by our adapted PCC_m), which are state-of-the-art and also discussed in this article (the former three as non-cache-based techniques and the last one as cache-based). They can all work with the preceding three scheduling strategies to form different combinations.
- *Checking workload.* As mentioned earlier, different groups of context changes contain different numbers of context changes as well as different field values, which will incur different checking workloads. As such, we used the 24 groups of context changes to evaluate and compare the performance of each strategy-technique combination.

Then, to evaluate the performance of each combination (required by the three research questions), we design two *dependent variables*:

- *Inconsistency missing rate.* It refers to the proportion of missed context inconsistencies (i.e., those not detected) against all inconsistencies in theory (using IMD as the baseline), which is for measuring the *quality* of context inconsistency detection (Section 2.5).
- *Checking time.* It refers to the amount of total time spent on checking context changes (including scheduling, checking, and any extra overhead), which is for measuring the *efficiency* of context inconsistency detection (Section 2.5).

Finally, we controlled some running environmental factors for facilitating the experiments. All experiments were conducted on a commodity PC with an Intel® Core™ i7-6700 CPU @3.41GHz with 16GB RAM and an Nvidia GTX 750 Ti card with 640 CUDA cores. The machine was installed with MS Windows 10 Professional and Oracle Java 8.

To answer RQ1 (motivation). We apply two scheduling strategies (IMD and BAT) to compare their performance. IMD schedules constraint checking upon each context change, and thus it serves as the oracle for detecting all context inconsistencies in theory. BAT schedules constraint checking upon every k context changes, where k is the batch size. This would improve BAT's efficiency by reduced scheduled checking, but also incur missed context inconsistencies. We would observe in the experiments how serious the inconsistency-missing situation is for BAT, and this would motivate our new proposal GEAS.

To answer RQ2 (effectiveness). We apply both GEAS's two versions (GEAS-ori and GEAS-opt), combined with the four constraint checking techniques (ECC, Con-C, GAIN, and PCC), to evaluate their performance, and compare it to that of the existing two scheduling strategies (IMD and BAT). GEAS should fulfill our targeted desirable scheduling strategy's two objectives, namely, quality and efficiency (Section 2.5). Regarding the quality objective, GEAS should never miss any context inconsistency in the detection, as contrast to BAT's high inconsistency missing rate. Regarding the efficiency objective, GEAS should be able to help all the four constraint checking techniques improve the detection efficiency, as compared to these techniques combined with IMD.

We first study how GEAS-ori is compared to IMD and BAT in preventing missing context inconsistencies and improving inconsistency detection efficiency. We then study the impact of batch size on BAT, i.e., how GEAS-ori is compared to BAT with various batch sizes. We next study the impact of constraint checking technique on GEAS-ori, i.e., how GEAS-ori is compared to IMD when it is combined with different constraint checking techniques. We also study the impact of checking workload on GEAS-ori, i.e., whether GEAS-ori's improvement on a constraint checking technique's inconsistency detection efficiency is consistent with various workloads. We finally compare GEAS-ori to GEAS-opt to see how the latter improves over the former, and also study its underlying reasons. The five studies together validate the effectiveness of our GEAS's two versions (GEAS-ori and GEAS-opt).

To answer RQ3 (overhead). Finally, we measure GEAS's time cost in its offline s-condition analysis and online s-condition matching/c-condition examination, to see whether the cost will compromise its effectiveness studied in RQ2. We study the time cost for GEAS's both versions (GEAS-ori and GEAS-opt).

5.3 Experimental Results and Analyses

In the following, we analyze the experimental results and answer the preceding three research questions in turn. For ease of presentation, we refer to a combination of a specific scheduling strategy x and a specific constraint checking technique y by the notation of x+y, e.g., IMD+ECC, BAT+PCC, and so on.

5.3.1 RQ1: Motivation

For research question RQ1, we conducted experiments to apply the IMD and BAT strategies to checking 6.75 million context changes in the SmartCity application. When combined with the IMD strategy, some techniques (e.g., ECC) are not so efficient, and can cost much more time than affordable by applications. For example, Fig. 8 shows IMD+ECC behaves with respect to 24 groups of hour-based context changes.

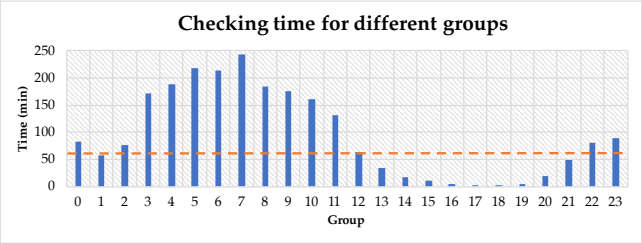


Fig. 8: Checking time for IMD+ECC with respect to 24 hour-based groups (the dashed line represents the one-hour limit for each group)

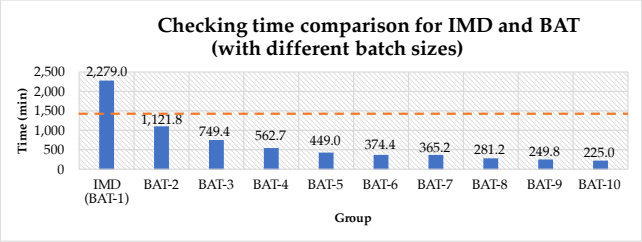


Fig. 9: Checking time comparison for the IMD and BAT strategies with respect to all 24-hour context changes (the dashed line represents the 24-hour limit)

In Fig. 8, we observe that the checking time varies greatly across different groups (from 1.5 minutes to 243.8 minutes, more than 100x difference), which is roughly proportional to the number of context changes in each group (in Fig. 7). This validates our earlier conjecture that the workload can directly affect the checking time of a specific constraint checking technique. For this example, what is worth noticing is that for many groups (14 out of 24, or 58.3%), the checking time has already exceeded the corresponding affordable limit, as the red dashed line illustrates in Fig. 8. The limit is set for illustration only, and its physical meaning is exactly one hour, exceeding which would suggest that checking all context changes takes even more time than that of collecting these changes, making such checking not meaningful in practice. Besides, the worst case is with group 7, which took IMD+ECC 243.8 minutes (more than four hours) to complete the checking, 306.3% exceeding the limit. Therefore, the IMD strategy is unacceptable when the workload is extremely heavy or the combined constraint checking technique is not that efficient. Considering various practical scenarios, this motivates for a better scheduling strategy that should be able to greatly reduce the checking time, as we claimed earlier.

For the comparison, Fig. 9 shows how BAT+ECC behaves with respect to all 24-hour context changes, when the BAT strategy is set with different batch sizes (from BAT-2 to BAT-10), and compares it to the IMD strategy, which is conceptually equivalent to BAT-1. To illustrate whether they have exceeded the 24-hour limit for all the context changes, we similarly use a red dashed line to represent the limit (similarly, exceeding this limit would suggest that such checking is not meaningful in practice). We observe that: (1) the IMD strategy took 2,279.0 minutes (about 38 hours) to complete the checking (58.3% exceeding the limit), (2) the BAT-2 strategy took only 1,121.8 minutes (about 19 hours)

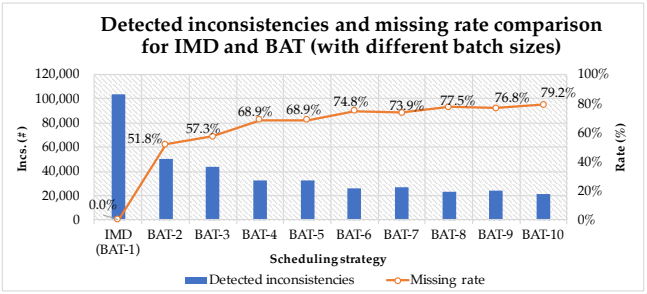


Fig. 10: Detected context inconsistencies and inconsistency missing rate comparison for the IMD and BAT strategies with respect to all 24-hour context changes

to complete, which already satisfies the limit and suggests BAT's immediate effectiveness on reducing the checking time, and (3) the BAT-x strategy took consistently decreasing checking time, with the growth of its batch size x, whose reduction rate can be up to 90.1% for BAT-10. However, the BAT strategy's seemingly promising effectiveness on reducing the checking time is accompanied with an uncomfortably high inconsistency missing rate, as Fig. 10 shows.

Fig. 10 compares the BAT strategy (with different batch sizes) to the IMD strategy on detected context inconsistencies and inconsistency missing rate with respect to all 24-hour context changes. We note that IMD's detected context inconsistencies serve as the oracle, as explained earlier (Section 5.2), for validating BAT's detected ones. In Fig. 10, we observe that: (1) with the growth of its batch size, the BAT strategy detected a nearly decreasing number of context inconsistencies, whose corresponding inconsistency missing rate keeps increasing and can be up to 79.2% for BAT-10, which is significant and suggests a clearly unacceptable result (most inconsistencies are missed), and (2) even if one sets the BAT strategy's batch size to its minimal value of 2, its corresponding inconsistency missing rate is still as high as 51.8%, suggesting that more than half of context inconsistencies are missed. From these observations, we conclude that the BAT strategy cannot be useful for constraint checking due to its high inconsistency missing rate, although it can help reduce the checking time.

Therefore, we answer research question RQ1 as follows:

Both the IMD and BAT strategies are undesirable. The former seriously limits the efficiency of context inconsistency detection, which the latter causes severe context inconsistency missing problems. This calls for a new scheduling strategy that can both improve the inconsistency detection efficiency and protect inconsistency detection results.

5.3.2 RQ2: Effectiveness

For research question RQ2, we compare our GEAS (GEAS-ori and GEAS-opt) with the IMD and BAT strategies on their context inconsistency detection efficiency and quality. We study the aforementioned five aspects, namely, selected scheduling strategy (GEAS-ori, IMD, or BAT), customized BAT's batch size (from 2 to 10), selected constraint checking technique (ECC, Con-C, GAIN, or PCC), varying checking workload (24 groups of context changes), and applied opti-

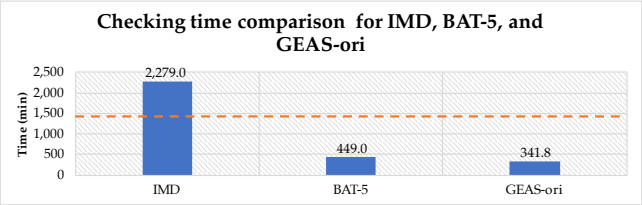


Fig. 11: Checking time comparison for the IMD, BAT-5, and GEAS-ori strategies (combined with the ECC technique) with respect to all 24-hour context changes (the dashed line represents the 24-hour limit)

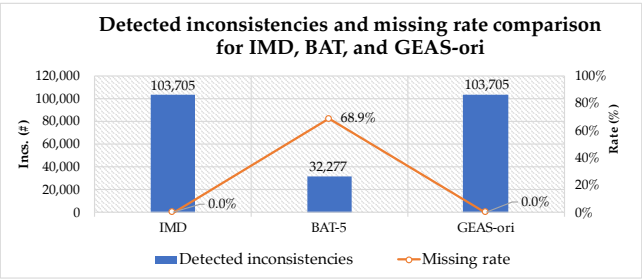


Fig. 12: Detected context inconsistencies and inconsistency missing rate comparison for the IMD, BAT-5, and GEAS-ori strategies (combined with the ECC technique) with respect to all 24-hour context changes

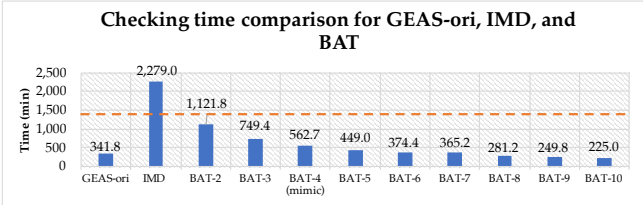


Fig. 13: Checking time comparison for three strategies (GEAS-ori, IMD, and BAT with different batch sizes, combined with the ECC technique) with respect to all 24-hour context changes (the dashed line represents the 24-hour limit)

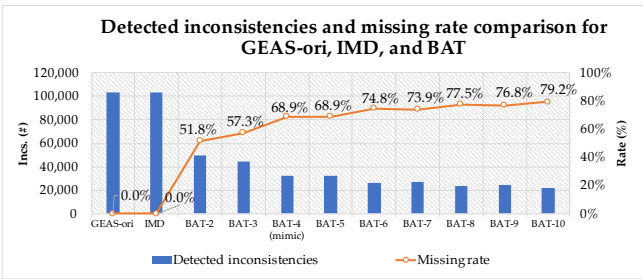


Fig. 14: Detected context inconsistencies and inconsistency missing rate comparison for three strategies (GEAS-ori, IMD, and BAT with different batch sizes, combined with the ECC technique) with respect to all 24-hour context changes

mization (GEAS-ori or GEAS-opt). We collect experimental results and analyze the five aspects in turn.

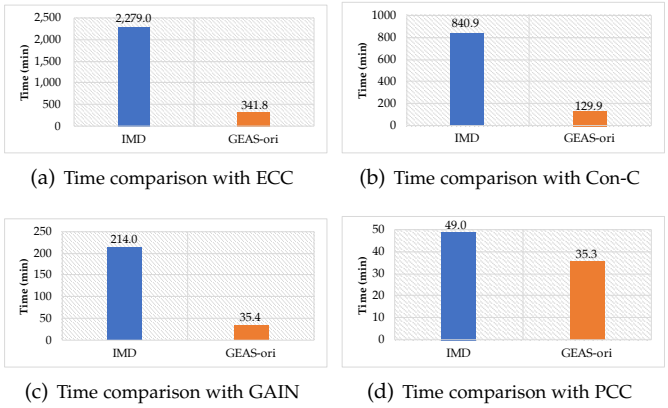
Aspect of selected scheduling strategy. We fix values of other independent variables to study how the selection of a specific scheduling strategy affects the efficiency and quality of context inconsistency detection. As such, we set the batch size to be 5 (half of its maximal value) for the BAT strategy, the combined constraint checking technique to be ECC, and the checking workload to be all 24-hour context changes as a whole. With this setup, we study the three scheduling strategies, namely, IMD, BAT-5, and our GEAS-ori, and compare their checking time and inconsistency detection results. We present the comparisons in Fig. 11 (on the checking time) and Fig. 12 (on the detected inconsistencies and missing rate).

In Fig. 11, we observe that IMD+ECC took the most time, 2,279.0 minutes, to complete checking all 6.75 million context changes. As mentioned earlier, this time cost already exceeds the 24-hour limit and is thus unacceptable. When using the BAT-5 strategy, it took only 449.0 minutes, with an 80.3% reduction to what IMD took. This is already below the limit and is promising. Nevertheless, when using the GEAS-ori strategy, the time cost was further reduced to 341.8 minutes, with an 85.0% reduction to that of IMD. These results suggest that both the BAT-5 and GEAS-ori strategies can help greatly reduce the checking time, corresponding to a large efficiency improvement of 407.6% and 566.7%, respectively.

We then compare BAT-5's and GEAS-ori's context inconsistency detection results and inconsistency missing rates in Fig. 12. As explained earlier, we used IMD+ECC's detected context inconsistencies (103,705) as the oracle for the comparison. We observe that: (1) the BAT-5 strategy detected only 32,277 inconsistencies, corresponding to a 68.9% inconsistency missing rate, which is quite frustrating; (2) on the contrary, the GEAS-ori strategy detected all 103,705 inconsistencies, suggesting a zero inconsistency missing rate, which is inspiring. Therefore, although the BAT-5 strategy (as a representative of the batch-based scheduling) can help improve the checking efficiency, its caused inconsistency missing problem is severe and unacceptable. On the contrary, our GEAS-ori strategy improves the efficiency even more, and at the same time protects each context inconsistency, i.e., all inconsistencies being successfully detected. This suggests that GEAS-ori is desirable, in terms of both our earlier analyzed efficiency and quality objectives.

Aspect of customized BAT's batch size. We then study how GEAS-ori is compared to BAT when the latter is customized with different batch sizes. Fig. 13 and Fig. 14 show the comparison results. From the two figures, we observe that with the growth of its batch size (from 2 to 10), BAT's checking time was consistently decreasing (from 49.2% to 9.9%, as compared to that of IMD), but at the same time caused a nearly increasing inconsistency missing rate (from 51.8% to 79.2%). These results are also shown in earlier Fig. 9 and Fig. 10. As a comparison, GEAS-ori took much less checking time (15.0%), and did not miss any context inconsistency in the detection (0% missing rate).

In the experiments, we are particularly interested in two comparisons: (1) GEAS-ori vs. BAT-7: BAT-7's checking time (365.2 minutes) is the closest to that of GEAS-ori (341.8 minutes). This indicates that BAT can be customized with a specific batch size so as to achieve a similar inconsistency detection efficiency improvement as GEAS-ori. However, their corresponding inconsistency missing rates differ greatly: 73.9% vs. 0%, and this indicates that when BAT is



(a) Time comparison with ECC (b) Time comparison with Con-C

(c) Time comparison with GAIN (d) Time comparison with PCC

Fig. 15: Checking time comparison for the IMD and GEAS-ori strategies (combined with the ECC, Con-C, GAIN, and PCC techniques, respectively) with respect to all 24-hour context changes

TABLE 2: Inconsistency missing rate comparison for the IMD and GEAS-ori strategies (combined with the ECC, Con-C, GAIN, and PCC techniques, respectively) with respect to all 24-hour context changes

Technique \ Missing rate	Scheduling strategy	
	IMD	GEAS-ori
ECC/Con-C/GAIN/PCC	0%	0%

customized close to GEAS-ori in efficiency improvement, BAT is actually useless due to its high missing rate. (2) GEAS-ori vs. BAT-4 (BAT-mimic): BAT-mimic is customized with a batch size equal to GEAS-ori's averaged batch size during its inconsistency detection. The averaged value is 3.58, to which the closest integer batch size is 4 for BAT. Thus we consider BAT-4 as BAT-mimic, which is supposed to be able to mimic GEAS-ori's behavior. However, we observe that BAT-mimic's checking time is 64.6% more than that of GEAS-ori, and that BAT-mimic's inconsistency missing rate is 68.9%, as compared to 0% of GEAS-ori. This indicates that when BAT is customized close to GEAS-ori in the batch size, BAT is both less effective in efficiency improvement and useless due to its high missing rate.

Combining the preceding comparisons and analyses, we conclude that: (1) no matter which batch size BAT is customized to, it cannot be compared to GEAS-ori; (2) GEAS-ori is effective in both efficiency improvement and inconsistency protection, not because of its batch-based scheduling nature, but because of its adaptive batch size control for separating different context changes.

Aspect of selected constraint checking technique. We next study the impact of a selected constraint checking technique on GEAS-ori, i.e., how GEAS-ori is compared to IMD on the efficiency and quality of context inconsistency detection when it is combined with different constraint checking techniques. Fig. 15 and Table 2 show the comparisons for the four constraint checking techniques (ECC, Con-C, GAIN, and PCC) on the checking time and inconsistency missing rate, respectively, for all 24-hour context changes.

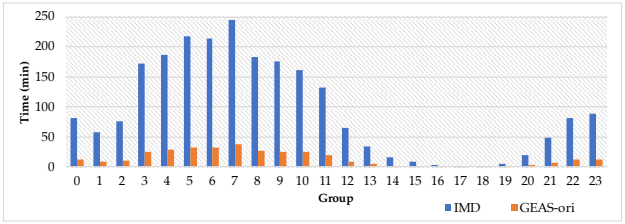
From Fig. 15, we observe that no matter which constraint checking technique is combined, the GEAS-ori strategy consistently largely reduced the checking time, as compared to the IMD strategy. The reduction rate is 85.0% for ECC, 84.6% for Con-C, 83.5% for GAIN, and 28.0% for PCC, which are significant. We note that the reduction rate seems not large for PCC (28.0%), but PCC works incrementally (different from the other three techniques) and is already the most efficient checking technique. For such an efficient technique, our GEAS-ori strategy can still help it reduce the checking time by 28.0%. As a result, GEAS-ori+PCC becomes the most efficient combination among all. Therefore, GEAS-ori can uniformly reduce the checking time for context inconsistency detection, as compared to IMD, although the reduction rate can vary with different constraint checking techniques.

TABLE 3: Efficiency improvement comparison (estimated vs. actual) for the GEAS-ori strategy, combined with three non-cache-based constraint checking techniques (ECC, Con-C, and GAIN)

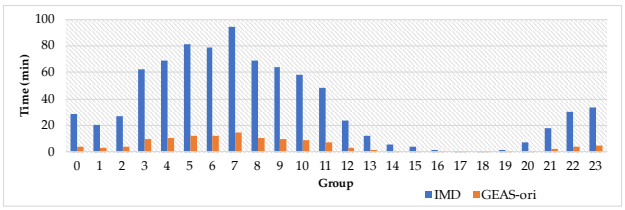
Checking technique	Scheduling (#)	Estimated efficiency improvement	Actual efficiency improvement
ECC	m: 2,216,748 (against n: 7,929,458)	+257.7% *	+566.7%
Con-C			+547.3%
GAIN			+504.5%

* calculated by $(n - m) / m$.

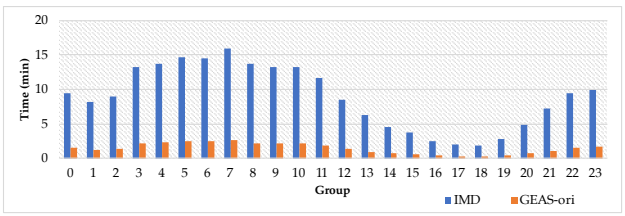
We then study the efficiency improvement of GEAS-ori against IMD from another perspective. Table 3 compares the efficiency improvement using our earlier estimated formula (i.e., $(n - m) / m$, in Section 2.5) and actual measurement, respectively, where n is the number of context changes, which equals to the number of scheduled constraint checking for IMD, and m is the number of scheduling for GEAS-ori, for the three non-cache-based constraint checking techniques (ECC, Con-C, and GAIN). We observe that for the three techniques, the estimated efficiency improvement is 257.7% (independent of the combined checking technique, only decided by the scheduling strategy), and that the actual efficiency improvement is 566.7%, 547.3%, and 504.5%, respectively. The latter is 1.96x–2.20x of the former. This result indicates that: (1) GEAS-ori indeed improves over IMD and its improvement extent is comparable to our estimation; (2) the improvement extent seems better than expected, and this is because the estimation assumes that each checking takes the same time (as analyzed earlier in Sections 2.5 and 3.5), but it may not necessarily hold for practical data. Regarding the cache-based constraint checking technique PCC, GEAS-ori uses its adapted version, i.e., PCC_m, for combination. As analyzed earlier in Section 3.5, each new change has to be checked, and when GEAS-ori reduces the number of PCC_m scheduling, PCC_m itself will have more changes to check in each checking (since it works incrementally). Thus GEAS-ori+PCC's efficiency improvement will not follow the estimation formula, but accumulating the checking results for multiple changes can now be done in one run instead of multiple runs, and thus GEAS-ori+PCC can still win IMD+PCC, i.e., 38.8% efficiency improvement.



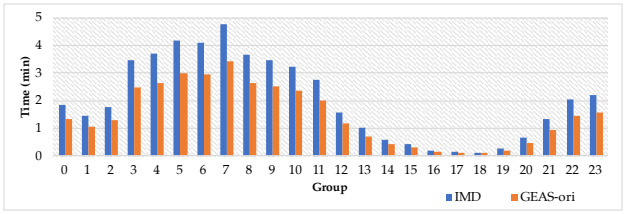
(a) Checking time comparison with ECC



(b) Checking time comparison with Con-C



(c) Checking time comparison with GAIN



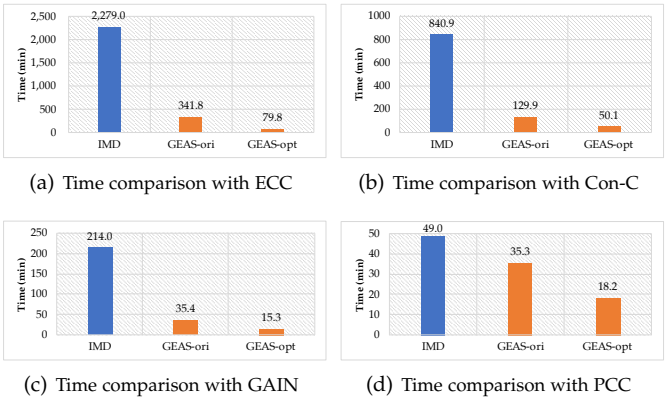
(d) Checking time comparison with PCC

Fig. 16: Checking time comparison for the IMD and GEAS-ori strategies (combined with the ECC, Con-C, GAIN, and PCC techniques, respectively) with respect to 24 hour-based groups

We also compare the inconsistency missing rate for the IMD and GEAS-ori strategies when combined with different constraint checking techniques in Table 2. From the table, we observe that no matter which constraint checking technique is combined, the GEAS-ori strategy consistently maintained zero inconsistency missing rate, as if no grouping of context changes occurred (i.e., as IMD did).

Combining the preceding comparisons and analyses, we conclude that no matter which constraint checking technique is combined, GEAS-ori can always improve the context inconsistency detection efficiency and at the same time protect all inconsistency detection results.

Aspect of varying checking workload. We next study the impact of the checking workload on the GEAS-ori strategy, i.e., whether GEAS-ori's improvement on a constraint checking technique's inconsistency detection efficiency is consistent with various workloads. We have 24 groups of context changes, which are naturally formed from 24 hours of raw taxi data, representing varying checking workloads,



(a) Time comparison with ECC

(b) Time comparison with Con-C

(c) Time comparison with GAIN

(d) Time comparison with PCC

Fig. 17: Checking time comparison for the GEAS-ori and GEAS-opt strategies (combined with the ECC, Con-C, GAIN, and PCC techniques, respectively) with respect to all 24-hour context changes (IMD data for reference)

TABLE 4: Inconsistency missing rate comparison for the GEAS-ori and GEAS-opt strategies (combined with the ECC, Con-C, GAIN, and PCC techniques, respectively) with respect to all 24-hour context changes (IMD data for reference)

Technique \ Missing rate	Scheduling strategy		
	IMD	**GEAS-ori**	**GEAS-opt**
ECC/Con-C/GAIN/PCC	0%	0%	0%

as discussed earlier in Section 5.2. Therefore, we observe how GEAS-ori is compared to IMD for different workloads when combined with the four constraint checking techniques (ECC, Con-C, GAIN, and PCC). Fig. 16 (a)–(d) show the comparisons on the checking time for the four techniques, respectively (we no longer compare the inconsistency missing rate since it is always zero, as studied in the last aspect).

From Fig. 16, we observe that different groups of context changes (representing different workloads) indeed incur greatly varying checking time (up to more than 100x difference), but the difference between GEAS-ori and IMD is basically consistent across different groups, no matter which constraint checking technique is combined: (1) with ECC, GEAS-ori achieved 80.3%–85.9% checking time reduction (average: 84.7%) against IMD across different groups; (2) with Con-C, the reduction is 68.5%–85.2% (average: 82.7%) for GEAS-ori vs. IMD; (3) with GAIN, the reduction is 82.8%–84.3% (average: 83.7%); (4) with PCC, the reduction is 17.3%–29.2% (average: 26.6%). Overall, the time reduction is around 80% for non-cache-based techniques (ECC, Con-C, and GAIN) and around 25% for the cache-based technique PCC, across different groups of context changes, with little variance, as illustrated in Fig. 16.

Therefore, we conclude that GEAS-ori's effectiveness on the efficiency improvement for context inconsistency detection is stable for different constraint checking techniques, even if its checking workload varies with different taxi data.

Aspect of applied optimization. Finally, we evaluate and compare GEAS-opt to GEAS-ori to show how the former additionally improves over the latter on the checking

TABLE 5: Time cost analyses

Checking time / Technique	For IMD	For GEAS-ori			For GEAS-opt		
	Online	Offline	Online		Offline	Online	
	T_{check} (min)	$T_{analysis}$ (ms)	T_{check} (min)	$T_{schedule}$ (min)	$T_{analysis}$ (ms)	T_{check} (min)	$T_{schedule}$ (min)
ECC	2,279.0	3.0	340.8	Around 1 min	3.0	69.8	Around 10 min
Con-C	840.9		128.9			40.1	
GAIN	214.0		34.4			5.3	
PCC	49.0		34.3			8.2	

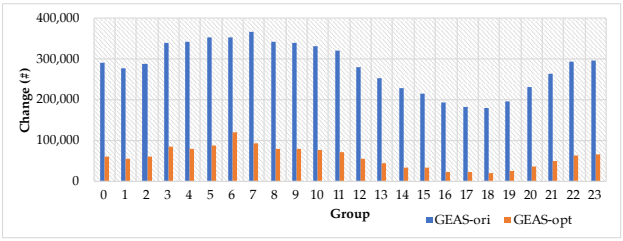
time (Fig. 17) and inconsistency missing rate (Table 4) in the context inconsistency detection for all 24-hour context changes.

From Fig. 17, we observe that GEAS-opt greatly reduces the checking time on top of GEAS-ori, no matter which constraint checking technique is combined. The time reduction is 76.7% for ECC, 61.4% for Con-C, 56.8% for GAIN, and 48.4% for PCC. Note that such efficiency improvements are achieved over what GEAS-ori has achieved. Therefore, taking IMD as the baseline, GEAS-ori's and GEAS-opt's efficiency improvement is 566.7% and 2,755.9% for ECC, 547.3% and 1,578.4% for Con-C, 504.5% and 1,298.7% for GAIN, and 38.8% and 169.2% for PCC. This shows GEAS's general effectiveness on improving the efficiency of context inconsistency detection, as well as GEAS-opt's additional benefits over GEAS-ori. Besides, from Table 4 we observe that GEAS-opt works as GEAS-ori, still keeping zero inconsistency missing rate, which is desirable, no matter which constraint checking technique is combined. This shows GEAS-opt, besides its additional efficiency improvement benefits, does not bring any negative consequence.

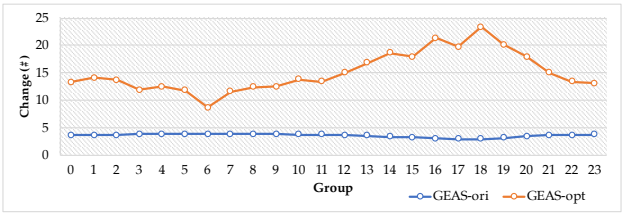
To understand GEAS-opt's superiority on efficiency improvement over GEAS-ori, we in the following further study two factors that relate to GEAS-opt's additional efficiency improvement: (1) number of context changes removed against GEAS-ori, and (2) batch size expanded against GEAS-ori.

First, as mentioned earlier, GEAS-opt improves over GEAS-ori by identifying and removing context change pairs that satisfy c-conditions of concerned constraints during context inconsistency detection. Fig. 18 (a) compares the numbers of context changes actually checked by GEAS-ori and GEAS-opt across 24 groups. We observe that GEAS-opt removed around 80.1% of context changes (varying from 66.4% to 89.1%), which is significant. This reduced workload directly contributes to GEAS-opt's efficiency improvement over GEAS-ori.

Second, GEAS-opt removes context changes satisfying c-conditions, thus refining batches of context changes for constraint checking, and if one counts these removed changes, GEAS-opt virtually increases its average batch size over GEAS-ori. Fig. 18 (b) compares the average batch sizes of GEAS-ori and GEAS-opt across 24 groups. We observe that GEAS-opt formed batches of size 13.7 (varying 8.7 to 23.4), 3.8x of that of GEAS-ori (size 3.6, varying 2.9 to 3.9). This increased batch size further explains why GEAS-opt largely improves over GEAS-ori on the inconsistency detection efficiency, since now much more context changes can be checked together.



(a) The distribution of context changes actually checked across different groups (difference between bars represents # removed changes by GEAS-opt)



(b) The distribution of the average (virtual) batch size across different groups

Fig. 18: Underlying reason analyses for GEAS-opt's superiority over GEAS-ori

Combining the preceding five aspects of comparisons. we answer research question RQ2 as follows:

The GEAS strategy is effective in improving the efficiency of context inconsistency detection and protecting inconsistency detection results. The effectiveness is consistent with respect to different constraint checking techniques, and stable for different checking workloads. Besides, GEAS-opt improves much more over GEAS-ori. Specially, GEAS-ori and GEAS-opt achieve 38.8%–566.7% (or 1.4x–6.7x) and 169.2%–2,755.9% (or 2.7x–28.6x) efficiency improvements over IMD, respectively, when combined with existing techniques (ECC, Con-C, GAIN, and PCC), and always maintain a zero inconsistency missing rate. GEAS also beats BAT due to its carefully controlled adaptive batch size, while the latter uses a fixed batch size and can miss up to 79.2% inconsistencies.

5.3.3 RQ3: Overhead

Finally, we analyze each part of the checking time to understand GEAS's overhead for supporting its effectiveness on the efficiency improvement against IMD. The checking time refers to the whole time cost of a strategy-technique combination in checking given 24-hour context changes. For the IMD strategy, it contains the online part only, i.e., the time used for checking all context changes (T_{check}). For

the GEAS strategy (GEAS-ori and GEAS-opt versions), it contains both the offline and online parts. The offline part refers to the time used for s-condition analysis ($T_{analysis}$), which is necessary only for GEAS (Section 5.2). The online part contains the conventional T_{check} (as IMD) and additional time used for scheduling constraint checking ($T_{schedule}$), as Algorithm 3 shows (Section 3.5). Therefore, GEAS's overhead is $T_{analysis} + T_{schedule}$. We study whether this overhead deserves GEAS's reduction on the whole time cost against IMD.

Table 5 lists each composed part of the checking time or whole time cost for the IMD and GEAS strategies when combined with four constraint checking techniques (ECC, Con-C, GAIN, and PCC). From this table, we observe that: (1) IMD's time costs (only T_{check}) are clearly more than those of GEAS's both versions ($T_{analysis} + T_{check} + T_{schedule}$), as we studied earlier; (2) the first part of GEAS's overhead ($T_{analysis}$) costs 3 milliseconds only, which is negligible for both GEAS-ori and GEAS-opt; (3) the second part of GEAS's overhead ($T_{schedule}$) costs around 1 minute and 10 minutes, respectively, for GEAS-ori and GEAS-opt, and occupies some portion of the whole time cost (GEAS-opt costs more time than GEAS-ori as it additionally examines c-conditions and updates concerned contexts, as explained in Section 4.4). Combining all parts together, we find that GEAS used an overhead of about 1 minute and 10 minutes, respectively, for GEAS-ori and GEAS-opt, but earned a reduction on the whole time cost of 13.7–1,937.2 minutes and 30.8–2,199.2 minutes, against IMD. We thus conclude GEAS's cost-effectiveness of trading this small overhead for much greater time reduction, achieving the overall efficiency improvement on context inconsistency detection.

We observe that $T_{schedule}$ seems large (around 10 minutes) to GEAS-opt as the major overhead. So we investigated its details, which include costs for the s-condition matching, c-condition examination, and batch forming and refining. The first part occupies one minute, similar to that in GEAS-ori. The second and third parts occupy about 6.4 and 2.6 minutes, respectively, forming an approximate ratio of 2.5 : 1. This ratio suggests that the c-condition examination took relatively more time for finding a context change in the current batch that has an opposite impact to the current change, while updating runtime trees (major step in the batch forming and refining) took less time, validating our earlier conjecture that the updating is efficient (Section 4.4). We further investigated the c-condition examination, and observed that in over 60% cases the context change having the opposite impact can be found at the first place in the batch, and when considering the first three places the percentage grows to over 75%. Considering that a batch has a size of 13.7 on average (Section 5.3.2), "first three" implies that the c-condition examination can find its target changes for cancellation quickly. This also suggests that GEAS-opt's optimization based on opposite-impact change cancellation can be easily conducted, since it does not require a pair of context changes to contain exactly equal elements.

Therefore, we answer research question RQ3 as follows:

GEAS's overhead is small and deserves its great efficiency improvement on context inconsistency detection. In particular, its time cost on the s-condition derivation is tiny and negligible.

TABLE 6: The selected three groups of context changes

Group (workload)	Period	Context change (#)	Average interval (ms)
Light	4am–5am	136,488	26.4
Median	8am–9am	280,556	12.8
Heavy	5pm–6pm	773,136	4.7

5.4 Case Study

In previous Sections (5.1–5.3), we have evaluated GEAS's performance by controlled experiments. We now evaluate GEAS's effectiveness in a case-study setting. We controlled a client thread to send context changes according to their exact timestamps to simulate the actual traffic conditions. We made a server client to receive the context changes and detect context inconsistencies in these changes with different strategy-technique combinations. Note that in previous controlled experiments, context changes were checked in turn and this process proceeded only when the last checking completed. As a result, no change or inconsistency was lost in the detection and we thus measured the actual time cost for checking all changes. As a comparison, in the case-study setting, context changes have to be checked according to their associated timestamps, and as a result changes might be missed or checked not at proper time points if some strategy-technique combinations are of low efficiency, leading to possible missed or wrongly reported context inconsistencies (i.e., false negatives or false positives). Therefore, this setting is more like the real world. By such measurement, we aim to evaluate GEAS's practical effectiveness in terms of detecting context inconsistencies and improving the detection efficiency.

Setup. We use the same SmartCity application but process a much larger dataset, which contains 11.53 million context changes (originally 6.75 million) for a continuous 24-hour day and 48 consistency constraints (originally 22) since it covers more vehicle types. We similarly partitioned the 11.53 million context changes into 24 groups (hours) and observed that these groups incur very different checking workloads. For comparison purposes, we selected three groups, representative as *light*, *median*, and *heavy* workloads, for the periods of 4am–5am (136,488 context changes), 8am–9am (280,556 changes), and 5pm–6pm (773,136 changes), respectively, as shown in Table 6. Consider that each period lasts one hour, their corresponding average intervals between sending every two consecutive changes are 26.4, 12.8, and 4.7 milliseconds, respectively, showing an increasing checking workload.

As previously evaluated, the BAT strategy has serious limitations of missing context inconsistencies (up to around 80%) in the detection even given enough checking time, and therefore we compare in the case study three scheduling strategies, namely, IMD, GEAS-ori, and GEAS-opt, all of which should not miss context inconsistencies in theory. For each of these strategies, we combine with them all four constraint checking techniques, namely, ECC, Con-C, GAIN, and PCC. As a result, we obtain a total of 12 strategy-technique combinations.

We measure three metrics, namely, *false negative rate*

TABLE 7: Case study results

Group (workload)	Checking technique	Oracle	For IMD			For GEAS-ori			For GEAS-opt		
		Inc (#)	Inc/ * (#)	T_{cost} (min)	R_{FN}/ R_{FP}	Inc/ * (#)	T_{cost} (min)	R_{FN}/ R_{FP}	Inc/ * (#)	T_{cost} (min)	R_{FN}/ R_{FP}
Light (4am–5am)	**ECC**	784	784	3.68	0%/0%	784	1.76	0%/0%	784	0.58	0%/0%
	Con-C		784	1.23	0%/0%	784	0.55	0%/0%	784	0.31	0%/0%
	GAIN		784	0.73	0%/0%	784	0.34	0%/0%	784	0.23	0%/0%
	PCC		784	0.22	0%/0%	784	0.20	0%/0%	784	0.17	0%/0%
Median (8am–9am)	**ECC**	6,912	565/ *390	61.82	94.4%/31.0%	6,912	17.98	0%/0%	6,912	4.40	0%/0%
	Con-C		6,912	20.36	0%/0%	6,912	7.44	0%/0%	6,912	3.22	0%/0%
	GAIN		6,912	11.18	0%/0%	6,912	4.33	0%/0%	6,912	2.09	0%/0%
	PCC		6,912	2.57	0%/0%	6,912	1.83	0%/0%	6,912	1.13	0%/0%
Heavy (5pm–6pm)	**ECC**	11,556	212/ *147	63.74	98.7%/30.6%	1,153/ *280	61.09	97.6%/75.7%	1,074/ *387	59.22	96.7%/64.0%
	Con-C		725/ *291	64.12	97.5%/60.0%	11,556	48.02	0%/0%	11,556	31.55	0%/0%
	GAIN		2,813/ *1,709	61.78	85.2%/39.2%	11,556	33.51	0%/0%	11,556	27.19	0%/0%
	PCC		6,095/ *5,483	59.80	52.6%/10.0%	11,556	29.43	0%/0%	11,556	26.22	0%/0%

* stands for the number of the true positives in the detection. If the datum after slash "/" is omitted, it means that all reported context inconsistencies are true positives.

(R_{FN}) for measuring the proportion of missed context inconsistencies against all context inconsistencies that should be reported, *false positive rate* (R_{FP}) for measuring the proportion of wrongly reported context inconsistencies against all reported context inconsistencies, and *time cost* (T_{cost}) for measuring the time each strategy-technique combination has actually spent on checking all context changes from a particular group. Note that in order to calculate the false negative/positive rates, we need an oracle of all context inconsistencies that should be reported, which has been obtained in a way as our earlier controlled experiments.

With the above context data, strategy-technique combinations, and designed metrics, we measured the performance of each of the 12 combinations for each of the three groups, and compare them in Table 7. Note that each measured time cost should not exceed 60 minutes theoretically for checking each group of context changes, but in practice even if the client stopped sending more changes, the server might still need slightly more time for clearing up remaining changes not processed yet due to specific low-efficiency combinations. Therefore, some few time cost data may slightly exceed 60 minutes.

Results. From Table 7, we make the following observations:

(1) The IMD scheduling strategy is undesirable because all constraint checking techniques combined with IMD can be subject to low-quality inconsistency detection results. Consider the most low-efficiency constraint checking technique ECC, although IMD+ECC produced satisfactory detection results (R_{FN} = 0% and R_{FP} = 0%) for the light workload, it suffered severe quality problems in the detection results for the median and heavy workloads (R_{FN} = 94.4%, 98.7%, and R_{FP} = 31.0%, 30.6%, respectively). Regarding techniques Con-C, GAIN, and PCC, since their efficiency is much higher than ECC, their combinations with IMD produced satisfactory detection results (all are R_{FN} = 0% and R_{FP} = 0%) even for the median workload. However, for the heavy workload, all the three combinations, namely, IMD+Con-C, IMD+GAIN, and IMD+PCC, still suffered different levels of quality problems in the detection results (R_{FN} = 97.5%, 85.2%, 52.6%, and R_{FP} = 60.0%, 39.2%, 10.0%, respectively). Therefore, we observe that when combined with IMD, none of the four constraint checking techniques can produce satisfactory results for all the three workloads. We owe this inability to the IMD scheduling strategy. This again justifies our motivation for a desirable scheduling strategy like GEAS.

(2) The GEAS scheduling strategy greatly improves the inconsistency detection results and detection efficiency. For all cases formed by a specific constraint checking technique and a specific workload (i.e., each line in Table 7, totally 12 cases except ECC+Heavy, which is discussed later), we observe that with GEAS against IMD: inconsistency detection results were all improved (all going for R_{FN} = 0% and R_{FP} = 0%); detection efficiency data were also all improved (up to about 1,300%); GEAS-opt always performed better than GEAS-ori (e.g., 12.2–308.6% higher efficiency). For the only exceptional case ECC+Heavy, because ECC is too low-efficiency, even with GEAS-ori and GEAS-opt, its inconsistency detection results were still unsatisfactory, although with improvement of some extent (R_{FN}: from 98.7%, to 97.6% and 96.7%; R_{FP}: from 30.6%, to 75.7% and 64.0%). We note that when the number of reported context inconsistencies is extremely small (less than 10% of the oracle data) due to the low-efficiency ECC, the R_{FP} values could be misleading. Therefore, to better interpret the data, we transform the two metrics into *true positives*. Then GEAS-ori+ECC and GEAS-opt+ECC, as compared to IMD+ECC,

increased the number of context inconsistencies that should be reported (i.e., real context inconsistencies) from 147 to 280 and 387, respectively. That is, when checking context changes with the most low-efficiency technique ECC, GEAS can still help detect 90.5% and 163.3% more real context inconsistencies.

(3) Low-efficiency constraint checking techniques, when combined with GEAS, may gain boosted performance even comparable to high-efficiency constraint checking techniques combined with IMD. For example, the most low-efficiency technique ECC, when combined with the IMD scheduling strategy, cost all possible time (61.82 minutes) but still missed 94.4% context inconsistencies with 31.0% wrongly reported results for the median workload. Nevertheless, when ECC is combined with the GEAS-opt strategy, it realized the perfect result of both 0% false negatives and 0% false positives, and at the same time cost only 4.40 minutes. This trivial time cost is even comparable to the combination of PCC (the most high-efficiency technique) and IMD (2.57 minutes). This suggests that the effectiveness of GEAS to low-efficiency constraint checking techniques can be amazing, although it has general effectiveness to all constraint checking techniques.

One may concern the possible delay GEAS has to take it order to group context changes before checking them as a whole. So we also measure and study such delay. First, we measured the average batch sizes for the three workloads (namely, light, median, and heavy), which are 4.0, 4.4, and 4.8 for GEAS-ori, and 55.2, 14.4, and 14.1 for GEAS-opt, respectively (meaning how many context changes in one batch on average). Here we note that the average batch sizes for GEAS-opt already include the numbers of canceled context changes for delay-computation purposes (their actual average batch sizes after cancellation are 2.8, 2.8, and 2.9). Then, we use the average intervals between every two context changes in Table 6 to estimate the delays (batch size $\times$ interval), which are 0.11 (GEAS-ori) and 1.46 (GEAS-opt) seconds for the light workload, 0.06 (GEAS-ori) and 0.18 (GEAS-opt) seconds for the median workload, and 0.02 (GEAS-ori) and 0.07 (GEAS-opt) seconds for the heavy workload. We can see that these delays are quite small. To further evaluate whether they are acceptable to respective workloads, we consider the average time periods required for detecting the next context inconsistency, which are 4.59, 0.52, 0.31 seconds for the three workloads, respectively, from Table 7. We observe that the delays are much smaller than the time periods for all the three workloads, and thus we believe that such delays are acceptable and worthwhile, considering that GEAS can bring along significant improvement on the quality and efficiency of inconsistency detection.

Therefore, we conclude that GEAS is also effective for practical context processing scenarios, by greatly improving the quality of inconsistency detection results and reducing the time costs.

5.5 Threat Analyses

One potential threat concerns the external validity of our experiments and conclusions, since we selected one taxi application as our subject. It is possible that the conclusions made on experiments with this application may not apply to other applications. Nevertheless, we have made necessary efforts to alleviate this threat. First, the SmartCity application and its accompanying consistency constraints and taxi data were mostly used in existing work [6], [10], [11], [12] on evaluating constraint checking techniques for their efficiency and quality. This facilitates our comparisons in the experiments for this work. Second, the consistency constraints and taxi data are realistic and representative for experiments, since the constraints cover all seven formula types in the constraint language and the data volume is huge (6.75 and 11.53 million context changes), greatly reducing the possibility of experimental bias due to special data. Third, the consistency constraints and taxi data together also incur different checking workloads for context inconsistency detection (more than 100x difference in the checking time), which are necessary for observing and comparing the performance of different strategy-technique combinations, as shown in our experiments. Fourth, to more comprehensively evaluate GEAS's performance on the application, constraints, and data, we conducted experiments in both controlled experimentation (Sections 5.1–5.3) and case study (Section 5.4) ways, validating GEAS's effectiveness in both controlled and practical environments. Therefore, we believe that these efforts can help alleviate as much as possible the potential threat to the external validity of our experimental conclusions.

Note that some experimental observations (e.g., specific comparison data) could be different for other applications. We argue that this only affects the degree of efficiency gains by GEAS over the immediate and batch-based scheduling strategies, but our experimental conclusions about GEAS's superiority over these strategies still hold, since we have formally proved GEAS's soundness by three theorems for guaranteeing its detection quality. Even if GEAS has its adaptive groups always being reduced to size one (almost impossible in practice), it conceptually becomes the immediate scheduling strategy. Therefore, GEAS's efficiency gain is almost always anticipated, considering its own marginal overhead.

Besides, to avoid possible implementation bias from affecting our experimental conclusions, we implemented or re-implemented all constraint checking techniques and scheduling strategies under study. We made them share the same data structures and file operation interfaces, rather than directly using their original implementations [6], [10], [11], [12]. We compared these implementations to their original ones for correctness with the help from the original authors. Moreover, other than these implementation efforts, we also formally specified our problem (Section 2.5) and proved its correctness by three theorems (Sections 3.2, 3.4, and 4.3). By doing so, we try our best towards the theoretically reliability of our experimental conclusions on GEAS's efficiency gain, detection quality, and generality.

Finally, we release the implementation[1] of our GEAS-ori and GEAS-opt at GitHub to facilitate readers to replicate our experiments. In addition, we also release a stand-alone tool[2] for deriving s-conditions for consistency constraints, so that readers can try our GEAS on other applications.

1. https://github.com/GenericAdaptiveScheduling/GEAS.
2. https://github.com/GenericAdaptiveScheduling/GenTool.

6 Discussions

In this section, we discuss some issues regarding GEAS's usage in practical application scenarios.

Which GEAS version to use. According to our evaluation results, we prefer GEAS-opt to GEAS-ori for a higher efficiency in context inconsistency detection. Nevertheless, there is one minor concern in making the selection for specific application scenarios. As aforementioned, GEAS-opt requires that target application scenarios have context addition and deletion changes only, while GEAS-ori does not have this restriction. Therefore, for an application scenario that has all three change types (i.e., addition, deletion, and update), we recommend either using GEAS-ori or using GEAS-opt with a special treatment to its received context update changes. The treatment would need to split each update change into one deletion change and one addition change, with a virtual lock on them. The virtual lock would require that: (1) the two changes should always be within the same batch, and (2) they are not allowed for cancellation. By doing so, we can extend the usage of GEAS-opt to application scenarios that also have context update changes, but this would incur some engineering effort, which we leave as future extension. Still, we made a preliminary analysis of application scenarios studied by major context inconsistency detection [6], [7], [8], [10], [11], [12], [38] and resolution work [19], [38], [39], [40], [41], [42], [43], [44], [45] to study their characteristics on using different types of context changes. We obtained a total of 15 unique application scenarios and found that most of them (ten) are directly applicable to GEAS-opt (i.e., no update change) and the remaining ones (five) can be easily adapted by the above treatment (i.e., transforming update changes into deletion and addition changes). Thus, we believe that our GEAS can apply to a wide range of application scenarios and the more efficient version GEAS-opt can be directly applicable in most cases.

Priorities of consistency constraints. GEAS has assumed that all consistency constraints are equally important and should report any of their violations. Under this assumption, GEAS tries to speed up the scheduling of constraint checking and protect the quality of inconsistency detection results. In practice, some constraints can have higher priorities than others in the sense that their violations must be reported and reported as soon as possible. For the former, such constraints work like "hard constraints" [6], which should not be violated in any case, and while for the latter, the constraints can be checked at later time when those of higher priorities have completed their checkings. GEAS can be configured to support such prioritized scheduling since it works at another level of controlling.

Handling of detection results. GEAS is responsible for supporting constraint checking techniques to more efficiently detect context inconsistencies. Regarding detected inconsistencies, there are various strategies for handling. Normally, each detected context inconsistency should be handled since it indicates a violation to a specific consistency constraint, which specifies some necessary property that should hold. Otherwise, the detected inconsistency may possibly lead to more severe consequences. This is supported by the study on the correlation between consistency error and application failure (67.1% from the former correlates to the latter, and 86.5% otherwise) [46]. Still, one may choose strategies of different emergency levels to handle detected inconsistencies, depending on the nature of consistency constraints or applications themselves [47], e.g., ignoring or tolerating inconsistencies (when the concerned constraint is soft [6]), or resolving inconsistencies by fixing concerned contexts [39], [40], [41], [42], [43], [44], [45]. In our SmartCity application, we detected quite a few context inconsistencies. We made a further study into these inconsistencies, and found that: (1) some are due to the sensitivity of some consistency constraints that restrict the calculation of traffic conditions to the major urban part of the concerned city and identify those contexts that are beyond this scope and thus can be easily removed from the calculation; (2) most are due to some consistency constraints from the application that monitor context changes on selected hot areas in an asynchronous way [7], whose detected inconsistencies are transient and can thus be tolerated within a short period of time as they will disappear automatically; (3) the remaining ones are real problems with the taxi data (e.g., localization errors due to GPS noise or lost signal), whose fixing needs application-specific resolution techniques. For example, consider one of the application's consistency constraints "all tracked taxis should be within the considered scope of the city", i.e., $\forall\ v \in C\ (($inScope$(v)))$. When it is violated, the inconsistency detection may report a link like (violated, $\{(v, \mathsf{t}_1)\}$). It indicates that a taxi with tracking record t_1 is now out of the considered scope of the city. Following the preceding first finding, the record t_1 can be removed from later use by the application. Consider another more complex constraint "all taxis should not drive beyond the maximal speed possible in the city (two consecutive tracking records for any taxi should not span across a large distance)", i.e., $\forall\ v_1 \in C_1\ (\forall\ v_2 \in C_2\ ($sameTaxi$(v_1, v_2)$ implies notLargeDist$(v_1, v_2)))$. When it is violated, the inconsistency detection may report a link like (violated, $\{(v_1, \mathsf{t}_1), (v_2, \mathsf{t}_2)\}$). It indicates that a taxi's two consecutive tracking records (t_1 and t_2, collected in a very short interval) induce a significantly large distance (more than the maximal possible speed). This suggests possible defects in the taxi's GPS data, which should be fixed. Following the preceding third finding, the inconsistency can be resolved by looking up earlier records before t_1 and later records after t_2 for this taxi, deciding which of t_1 and t_2 are more likely to be faulty, and then replacing it with an interpolated point derived from this series of records, or by other application-specific resolution techniques. We shall give a detailed treatment of such resolution techniques in the next section.

7 Related Work

This work focuses on quality assurance for adaptive applications. As mentioned earlier in Section 1, such applications increasingly rely on their interactions with environments to decide when and how to adapt to environmental changes for delivering smart services. Typical examples of these applications include early ConChat [48], ActiveCampus [49], 3dID [50], Locale [51], and latest self-driving vehicles [1], [2], [3]. The quality assurance for such applications depends much on the whole interaction loop between an

application and its environment, which concerns how to obtain high-quality raw sensory data, how to guarantee the consistency of application contexts based on other raw data (e.g., sensory data), how to identify potential context-related adaptation defects in applications, as well as how to guarantee the consistency of general software artifacts in the application development. In the following, we present and discuss representative related work along these four lines in recent years.

Obtaining high-quality raw sensory data. Raw sensory data can be similar to application contexts from the point of view of their structures, but the former are typically processed by data engineering techniques. These techniques focus mainly on cleansing noisy data by pre-specified filtering thresholds or adaptive tuning, such that the cleansed data can meet certain quality requirements. For example, the RFID technology [27] is being widely used for object identification and tracking, but it often suffers from missing or cross read problems [28], [29], [30], [31], which refer to an object located in the RFID sensing range being undetected due to metal material or human body absorption, or an object being misread due to misconfigured voltage or misplaced antennas. Such problems are often referred to as RFID data anomalies [29], and can be removed by various techniques based on filtering [28], fuzzy operators [52], or sequence-based rules [29]. They may also be corrected by measuring collected data's probabilities of being correct based on integrated constraints and if necessary replacing them with new versions based on entropy maximization [53]. Some techniques have also been integrated into commercial ETL products [54], [55] for automated RFID data cleansing. However, one major limitation of these techniques is that they focus mainly on setting up or tuning filtering thresholds that concern characteristics of certain types of sensory data, and thus are unaware of application requirements on these data (e.g., assuring the consistency among several different types of sensory data and alleviating the impact of cleansing sensory data on application behavior).

Guaranteeing the consistency of application contexts. Application contexts can be derived from sensory data (e.g., location and temperature data), collected from user or application profiling data (e.g., user calendar and executed task information), or learned from user behavior or real-world events (e.g., user mood and activity). Due to their multi-source and various-processing nature, application contexts can be inaccurate, incomplete, or even conflicting with each other, known as context inconsistency. As a result of such complexity, threshold or tuning based data engineering techniques may not suffice to address. This calls for efforts of comprehensive context management techniques.

Existing context inconsistency detection techniques can be classified into two approaches. One approach uses dedicated rules for specifying inconsistency scenarios an application cannot tolerate, and then detects the occurrences of such scenarios. For example, Bu et al. [41], [56] proposed an ontology-based context model, from which rules are specified for describing inconsistency scenarios so as to facilitate their detection. Xu et al. [38] proposed semantic matching and inconsistency triggers to specify inconsistency scenarios, and integrating them into a middleware infrastructure, Cabot, for supporting inconsistency-free context-aware computing. Due to the difficulty of enumerating almost infinite inconsistency scenarios, the other approach specifies what desirable scenarios are in term of consistency constraints, and judges any violation of such specification as an occurrence of context inconsistency. For example, existing work proposed various techniques to detect context inconsistencies and identify problematic contexts based on the notion of consistency constraint, by means of full constraint checking (ECC, integrated in the *xlinkit* tool) [8], incremental constraint checking (PCC) [6], CPU-parallel constraint checking (Con-C) [10], and GPU-parallel constraint checking (GAIN) [11]. We observe an increasing trend of recognizing the necessity of efficiency detection for context inconsistency, due to the fundamental difference between application contexts, which are subject to frequent changes, and other traditional software artifacts, which are usually static or change rarely or slowly. Besides, almost all existing work on context inconsistency detection focuses on speeding up constraint checking, except our work (its preliminary version [12] and extended version in this article) is dedicated for reducing the scheduling, i.e., removing unnecessary constraint checking. Regarding scheduling, there are also some pieces of related work in other fields, e.g., scheduling abstraction refinement for verifying sequential consistency for concurrent programs [57], scheduling towards maximal throughput for network applications [58], [59], scheduling tasks for parallel and distributed heterogeneous computing systems [60], [61], [62], and scheduling predicate evaluations in distributed environments upon coming events [63], [64]. These pieces of work do not focus on reducing unnecessary tasks for scheduling so as to improve the efficiency, different from the focus of our work in this article.

Detected context inconsistencies can be resolved by identifying and processing the contexts that are responsible for these inconsistencies, which are known as inconsistent contexts. Existing context inconsistency resolution techniques can also be classified into two approaches. One approach proposes resolving context inconsistencies from the context perspective only, i.e., processing inconsistent contexts as if they were pure data, without considering any side effect of their processing to applications that use the contexts. For example, Ranganathan et al. [39] and Insuk et al. [40] proposed following human preferences to identify and process inconsistent contexts. This technique is simple and easy to implement, but may not apply to dynamic scenarios whose requirements are subject to change. Bu et al. [41] suggested considering all contexts relating to any context inconsistency as inconsistent ones, and removing all of them except the latest ones, assuming that the latest contexts have the most reliability. This technique is based on heuristics, which works efficiently but may remove more than necessary contexts, leading to context losses. Chomicki et al. [42] considered one randomly selected event/action to be responsible for removal so as to resolve the conflicts among multiple events/actions. This technique can directly apply to the context inconsistency resolution problem. These discussed techniques would accidentally change available contexts to applications, thus affecting application behavior unexpectedly. The other approach explicitly considers this problem and tries to alleviate its impact. For example, Xu et al. [45] considered those contexts that participate more

frequently in inconsistencies to be more likely inconsistent. One advantage of this technique is that it can minimize the number of contexts that have to be removed, indirectly alleviating potential side effect to applications that run with these contexts. Xu et al. [43], [44] later proposed explicitly quantifying the side effect that can be caused by any context-processing step, and selecting the resolution strategy that minimizes the summation of all possible side effects caused by the steps in this strategy. These techniques attempt to alleviate possible side effect of context inconsistency resolution to applications. However, since their main focus is on inconsistency resolution, these techniques still leave a large room for applications to possibly suffer from other problems (e.g., adaptation defects) due to the lack of sufficient consideration of processing various types of contexts.

Identifying context-related adaptation defects. Contexts can be of various types (e.g., sensory data, user profiles, or derived information) and of different qualities (e.g., inconsistent or resolved). This complexity makes developers easily fail to sufficiently consider proper logics of processing the contexts and delivering smart services in a context-aware way. Thus incurred problems are often referred to as *context-related adaptation defects*. Some pieces of work focus on verifying or monitoring general properties identified in context-aware adaptive applications, e.g., predictability [4], [26], [32], [39], [40], [42], [65], stability [26], [32], [51], [65], reachability and liveness [32], and consistency [66]. Their violation indicates the presence of adaptation defect, which can drive an application to behave abnormally, e.g., application behavior no longer predictable upon any situation, application no longer stable after any adaptation, application state no longer reachable or adaptation no longer triggerable, and application's perception to its environment no longer consistent with its actual environmental conditions [23]. Other pieces of work assume the availability of specific assertions for judging whether a context-aware adaptive application has gone into an abnormal state (e.g., crashing, taking illegal operations, and GUI freezing), and use model checking or testing techniques to check the satisfiability of such assertions. For example, Yang et al. [67], [68], [69] proposed combining path conditions with environmental/uncertainty constraints and solving them by model checking for counterexamples that violate the given assertions. The generated counterexamples then give possible scenarios where abnormal adaptation can occur. Wang et al. [70], Lu et al. [71], and Lai et al. [72] tried to improve the testing adequacy for context-aware adaptive applications by either strengthening testing coverage or introducing new test coverage criteria. These pieces of work aim for identifying defects in an application's adaptation logics in processing contexts of various types and qualities, complementing existing research efforts on context inconsistency detection and resolution, thus assuring quality for adaptive applications from both inside software and outside software.

Guaranteeing the consistency of general software artifacts. The preceding three lines of related work all focus on the runtime of adaptive applications, i.e., after the applications are deployed. This concerns one important phase of the whole software development. In fact, the software engineering community has extensively studied the consistency management issue for general software artifacts generated in the whole software development process, and the proposed techniques share some degree of similarity. For example, various techniques have been proposed for managing the consistency of many traditional software artifacts, e.g., XML documents [8], [9], [14], UML models [15], [16], [17], data structures [18], workflows [19], and distributed source code [20]. One feature of these artifacts is that they are manually made or indirectly affected by human operations, thus subject to various inconsistency issues.

However, as mentioned earlier, these software artifacts are typically static or change rarely or slowly, and thus these techniques focus mainly on the effectiveness of consistency management. Context, the artifact studied in this article, is featured by its dynamics and imperfectness, and thus requires for a special treatment on the efficiency of its consistency management (e.g., PCC [6], Con-C [10], and GAIN [11]). Besides, context consistency management has to address an additional challenge that traditional consistency management typically overlooks, i.e., deciding the proper boundaries for deciding when to schedule constraint checking for inconsistencies, since contexts are typically formed by infinite streams. If the boundaries are decided wrongly, reported inconsistencies can be meaningless or misleading.

In fact, this issue has also be partially observed in the consistency management of traditional software artifacts, e.g., editing scripts from human users. If constraint checking is scheduled at wrong time points, incorrectly grouped editing scripts can incur numerous false warning [73], unexpectedly frustrating users. However, this issue has never become a major focus in the consistency management of traditional software artifacts, but for context inconsistency detection, it can be critical if not addressed properly. For example, Xu et al. confirmed common occurrences of unstable context inconsistencies (one type of false warning) in context inconsistency detection, and proposed proactively suppressing them by deriving instability conditions [7]. Later, Xi et al. proposed pattern learning and dynamic matching in context streams to adaptively suppress inconsistency hazards (another type of false warning) [74].

The work studied in this article takes one step further by exploiting the implicit relationship between improving the efficiency for context inconsistency detection and suppressing undesirable results (e.g., false warning) in the detection. Our GEAS transforms the isolation of normal inconsistencies from undesirable results into the identification of critical time points when scheduling constraint checking can potentially cause missed inconsistencies, and then exploits such information to decide when the scheduling can or cannot be skipped (i.e., grouping concerned contexts or not), so as to improve the efficiency for context inconsistency detection. As we show in the evaluation, the efficiency improvement can be significant.

8 Conclusion

In this article, we study the context inconsistency detection problem, whose major concern lies on its detection efficiency and quality. Existing research efforts either have limited the detection efficiency since they commonly assume the use of the immediate scheduling strategy for combination,

or negatively affected the detection quality when trying to combine with the batch-based scheduling strategy.

To address this efficiency-quality dilemma, we have proposed a novel scheduling strategy GEAS with carefully designed mechanisms to improve the detection efficiency with quality guarantee. We design GEAS with two insights. First, GEAS uses s-conditions to break latent interferences between collected context changes and group them adaptively with batches of dynamic sizes. This mechanism improves the detection efficiency without any missing of inconsistency results. Second, GEAS additionally uses c-conditions to remove impact-opposite context changes from formed batches. This mechanism optimizes the workload by conducting safe change removal and thus further improves the detection efficiency. We have formally proved the correctness of these two mechanisms and successfully applied them to existing four constraint checking techniques, showing GEAS's generality. The experimental evaluation shows that GEAS can largely improve the efficiency for context inconsistency detection by 38.8%–566.7% (or 1.4x–6.7x) for GEAS-ori and 169.2%–2,755.9% (or 2.7x–28.6x) for GEAS-opt without any missing of inconsistency results. When applied to a practical application scenario, GEAS exhibited both significant efficiency improvement and unique superiority over other scheduling strategies on the quality of inconsistency detection results, no matter which constraint checking technique is combined with.

Future directions. Our GEAS still has some limitations and deserves further research along this line. First, our further study shows that although GEAS have greatly suppressed unnecessary scheduling of constraint checking (e.g., GEAS-ori: by 72.0%, GEAS-opt: by 92.7%), we still observed some scheduling being free of any context inconsistency result, i.e., potential room for further improvement. Our s-conditions statically derive those time points when improper scheduling constraint checking can miss inconsistency results, and are conservative due to its lack of dynamic information. Our c-conditions partially add such dynamic information by checking whether certain change pairs can cause opposite impacts in constraint checking, but its checking is for changes associated with the same contexts only. Additional checking for changes across different contexts can be possible but would be much more complicated, and this could be a direction that deserves efforts for further efficiency improvement of context inconsistency detection.

Second, we studied the application of GEAS to centralized massive data processing for context-aware adaptive applications. In fact, GEAS can also work for resource-restricted computing devices, e.g., mobile phones or wireless sensor networks where computations are pervasive but environmental data are dynamic and noisy. However, GEAS's application to distributed scenarios, although promising, deserves further research about its feasibility. Our previous work [75] explored the possibility of decentralized constraint checking, but was found to be communication-intensive. Therefore, it deserves investigating whether GEAS-alike techniques can help suppress unnecessary communications across distributed hosts, but still maintain complete inconsistency detection results.

Third, now context inconsistency detection has various scheduling strategies (e.g., IMD, BAT, GEAS-ori, and GEAS-opt) and constraint checking techniques (e.g., ECC, Con-C, GAIN, and PCC). Application developers and users can have a variety of combinations to choose from. Then one natural question arises: how should one choose the most suitable one from all possible combinations, considering that these combinations have various efficiency levels and space/time requirements? A further question is: since an application scenario can be subject to change in terms of checking workload, e.g., the taxi scenario has both rush hours with heavy traffic and silent midnights without much traffic, then how can one switch from one strategy-technique combination to another without affecting inconsistency detection results by continually, non-stopping constraint checking? Such questions deserve further investigations when one plans to deploy various strategy-technique combinations to practical application scenarios.

ACKNOWLEDGMENTS

The authors wish to thank the editor and anonymous reviewers for their valuable comments on improving this article. This work was supported in part by National Key R&D Program (Grant #2017YFB1001801) and National Natural Science Foundation (Grant #61690204) of China. The authors would also like to thank the support of the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

REFERENCES

[1] "California DMV. 2016. autonomous vehicle disengagement reports," 2016. [Online]. Available: https://www.dmv.ca.gov/portal/dmv/detail/vr/autonomous/disengagement_report_2016

[2] "The numbers don't lie: Self-driving cars are getting good." 2017. [Online]. Available: https://www.wired.com/2017/02/california-dmv-autonomous-car-disengagement

[3] "Autonomous vehicles enacted legislation," 2017. [Online]. Available: http://www.ncsl.org/research/transportation/autonomous-vehicles-self-driving-vehicles-enacted-legislation.aspx.

[4] 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.

[5] A. H. van Bunningen, L. Feng, and P. M. G. Apers, "Context for ubiquitous data management," in *Proceedings of the 2005 International Workshop on Ubiquitous Data Management (UDM)*, Tokyo, Japan, April 2005, pp. 17–24.

[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, 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.

[8] 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.

[9] S. P. Reiss, "Incremental maintenance of software artifacts," in *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM)*, Sep 2005, pp. 113–122.

[10] 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.

[11] 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.

[12] B. Guo, H. Wang, C. Xu, and J. Lu, "GEAS: Generic adaptive scheduling for high-efficiency context inconsistency detection," in *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Oct 2017, pp. 137–147.

[13] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Proactive detection of collaboration conflicts," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE)*. ACM, 2011, pp. 168–178.

[14] 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.

[15] 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.

[16] 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.

[17] "Argouml," http://argouml.tigris.org/.

[18] 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.

[19] 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.

[20] 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.

[21] D. Salber, A. K. Dey, and G. D. Abowd, "The context toolkit: aiding the development of context-enabled applications," in *ACM Conference on Human Factors in Computing Systems (CHI)*, Pittsburgh, PA, USA, May 1999, pp. 434–441.

[22] T. H. Tse, S. S. Yau, W. K. Chan, H. Lu, and T. Y. Chen, "Testing context-sensitive middleware-based software applications," in *Proceedings of the 28th Annual International Computer Software and Applications Conference*, Sep 2004, pp. 458–466.

[23] 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.

[24] C. L. K. L. C. Xu, S.C. Cheung and J. Wei, "Cabot: On the ontology for the middleware support of context- aware pervasive applications," in *Proceedings of the IFIP Workshop on Building Intelligent Sensor Networks*, Otc 2004, pp. 568–575.

[25] 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.

[26] M. Sama, D. S. Rosenblum, Z. Wang, and S. Elbaum, "Multilayer 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.

[27] R. Want, "RFID: A key to automating everything," *Scientific American*, vol. 290, no. 1, pp. 56–65, 2004.

[28] S. R. Jeffery, M. Garofalakis, and M. J. Franklin, "Adaptive cleaning for RFID data streams," in *Proceedings of the 32nd International Conference on Very Large Data Bases (DBLP2006)*, Seoul, Korea, September 2006.

[29] J. Rao, S. Doraiswamy, H. Thakkar, and L. S. Colby, "A deferred cleansing method for RFID data analytics," in *Proceedings of the 32nd International Conference on Very Large Data Bases (DBLP2006)*, Seoul, Korea, September 2006, pp. 175–186.

[30] K. T. Patil, V. Bansal, V. Dhateria, and S. K. Narayankhedkar, "Probable causes of RFID tag read unreliability in supermarkets and proposed solutions," in *International Conference on Information Processing*, 2016, pp. 392–397.

[31] N. Fescioglu-Unver, S. H. Choi, D. Sheen, and S. Kumara, "RFID in production and service systems: Technology, applications and issues," *Information Systems Frontiers*, vol. 17, no. 6, pp. 1369–1380, 2015.

[32] 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.

[33] E. Guizzo, *Three Engineers, Hundreds of Robots, One Warehouse*. IEEE Press, 2008.

[34] A. Rosenfeld, "Human-multi-robot team collaboration using advising agents: (doctoral consortium)," in *Proceedings of the 2016 International Conference on Autonomous Agents and Multi-agent Systems*, ser. AAMAS '16, Richland, SC, 2016, pp. 1516–1517.

[35] "STO express sorting system," last accessed in Jan 2018. [Online]. Available: http://www.sohu.com/a/133167323_700653

[36] "These robots sort packages efficiently," last accessed in Jan 2018. [Online]. Available: https://www.msn.com/en-gb/video/viral/these-robots-sort-packages-incredibly-efficiently/vi-BBzF7BL

[37] C. Xu, S. C. Cheung, and W. K. Chan, "Goal-directed context validation for adaptive ubiquitous systems," in *International Workshop on Software Engineering for Adaptive and Self-Managing Systems, 2007. ICSE Workshops SEAMS*, Washington, DC, USA, 2007, p. 17.

[38] 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 (ESEC/FSE)*, vol. 30, no. 5. Lisbon,Portugal: ACM, 2005, pp. 336–345.

[39] 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.

[40] P. Insuk, D. Lee, and S. J. Hyun, "A dynamic context-conflict management scheme for group-aware ubiquitous computing environments," in *Proceedings of the 29th Annual International Computer Software and Applications Conference - Volume 01*, ser. COMPSAC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 359–364.

[41] Y. Bu, T. Gu, X. Tao, J. Li, S. Chen, and J. Lu, "Managing quality of context in pervasive computing," in *Proceedings of the Sixth International Conference on Quality Software*, ser. QSIC '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 193–200.

[42] J. Chomicki, J. Lobo, and S. Naqvi, "Conflict resolution using logic programming," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 15, pp. 244–249, 01 2003.

[43] C. Xu, X. Ma, C. Cao, and J. Lu, "Minimizing the side effect of context inconsistency resolution for ubiquitous computing," in *Mobile and Ubiquitous Systems: Computing, Networking, and Services*, A. Puiatti and T. Gu, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 285–297.

[44] 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 (ESEC/FSE)*, Dubrovnik, Croatia, Sep 2007, pp. 569–572.

[45] ——, "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.

[46] C. Xu, W. Yang, X. Ma, and C. Cao, "Environment rematching: toward dependability improvement for self-adaptive applications," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Press, 2013, pp. 592–597.

[47] A. Russo, S. Easterbrook, and B. Nuseibeh, "Leveraging inconsistency in software development," *Computer*, vol. 33, pp. 24–29, 04 2000. [Online]. Available: doi.ieeecomputersociety.org/10.1109/2.839317

[48] 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.

[49] 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.

[50] 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*, ser. DATE '06. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2006, pp. 136–141.

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

[52] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani, "Robust and efficient fuzzy match for online data cleaning," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of*

Data, ser. SIGMOD '03. New York, NY, USA: ACM, 2003, pp. 313–324.

[53] N. Khoussainova, M. Balazinska, and D. Suciu, "Towards correcting input data errors probabilistically using integrity constraints," in *Proceedings of the 5th ACM International Workshop on Data Engineering for Wireless and Mobile Access*, ser. MobiDE '06. New York, NY, USA: ACM, 2006, pp. 43–50.

[54] "Informatica," http://www.informatica.com/.

[55] "Ibm websphere quality stage," http://ibm.ascential.com/products/websphere_qualitystage.html.

[56] Y. Bu, S. Chen, J. Li, X. Tao, and J. Lu, "Context consistency management using ontology based model." *Lecture Notes in Computer Science*, vol. 4254, pp. 741–755, 2006.

[57] L. Yin, W. Dong, W. Liu, and J. Wang, "Scheduling constraint based abstraction refinement for weak memory models," in *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Montpellier, France, Sep 2018, pp. 645–655.

[58] P. Chaporkar and A. Proutiere, "Adaptive network coding and scheduling for maximizing throughput in wireless networks," in *Proceedings of the 13th annual ACM international conference on Mobile computing and networking*. ACM, 2007, pp. 135–146.

[59] L. Tassiulas and A. Ephremides, "Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks," *IEEE transactions on automatic control*, vol. 37, no. 12, pp. 1936–1948, 1992.

[60] E. S. Hou, N. Ansari, and H. Ren, "A genetic algorithm for multiprocessor scheduling," *IEEE Transactions on parallel and distributed systems*, vol. 5, no. 2, pp. 113–120, 1994.

[61] Y. Xu, K. Li, J. Hu, and K. Li, "A genetic algorithm for task scheduling on heterogeneous computing systems using multiple priority queues," *Information Sciences*, vol. 270, no. 6, pp. 255–287, 2014.

[62] Y. T. Leung, "A new algorithm for scheduling periodic, real-time tasks," *Algorithmica*, vol. 4, no. 1–4, pp. 209–219, 1989.

[63] A. D. Kshemkalyani, "Temporal predicate detection using synchronized clocks," *IEEE Transactions on Computers*, vol. 56, no. 11, pp. 1578–1584, 2007.

[64] ——, "Immediate detection of predicates in pervasive environments," *Journal of Parallel and Distributed Computing*, vol. 72, no. 2, pp. 219–230, 2012.

[65] B. H. C. Cheng, R. D. Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, and B. Cukic, "Software engineering for self-adaptive systems: A research roadmap," in *Software Engineering for Self-Adaptive Systems*, 2009, pp. 1–26.

[66] D. Kulkarni and A. Tripathi, "A framework for programming robust context-aware applications," *IEEE Transactions on Software Engineering*, vol. 36, no. 2, pp. 184–197, 2010.

[67] W. Yang, C. Xu, Y. Liu, C. Cao, X. Ma, and J. Lu, "Verifying self-adaptive applications suffering uncertainty," in *ACM/IEEE International Conference on Automated Software Engineering*, 2014, pp. 199–210.

[68] W. Yang, C. Xu, M. Pan, C. Cao, X. Ma, and J. Lu, "Efficient validation of self-adaptive applications by counterexample probability maximization," *Journal of Systems and Software*, vol. 138, pp. 82–99, 2018.

[69] W. Yang, C. Xu, M. Pan, X. Ma, and J. Lu, "Improving verification accuracy of CPS by modeling and calibrating interaction uncertainty," *ACM Transactions on Internet Technology*, vol. 18, no. 2, pp. 1–37, 2018.

[70] Z. Wang, D. S. Rosenblum, and S. Elbaum, "Automated generation of context-aware tests," in *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, vol. 00, Washington, DC, USA, 05 2007, pp. 406–415.

[71] 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.

[72] Z. Lai, S. C. Cheung, and W. K. Chan, "Inter-context control-flow and data-flow test adequacy criteria for nesc applications," in *ACM Sigsoft International Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2008, pp. 94–104.

[73] 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.

[74] W. Xi, C. Xu, W. Yang, X. Ma, P. Yu, and J. Lu, "Suppressing detection of inconsistency hazards with pattern learning," *Information and Software Technology (IST)*, vol. 74, pp. 219 – 229, 2016.

[75] C. Xu and S. C. Cheung, "Decentralized constraint checking for pervasive computing," in *Proceedings of the 6th Annual IEEE International Conference on Pervasive Computing and Communications (Ph. D. forum), Hong Kong*, 2008, pp. 45–48.

Huiyan Wang is a PhD student with the Department of Computer Science and Technology at Nanjing University, China. She received her BSc degree in computer science and technology from Nanjing University in 2015. Her research interests include automated software engineering, software testing and analysis, and software methodologies.

Chang Xu received his doctoral degree in computer science and engineering from The Hong Kong University of Science and Technology, Hong Kong, China. He is a full professor with the State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology at Nanjing University. He participates actively in program and organizing committees of major international software engineering conferences. He co-chaired the MIDDLEWARE 2013 Doctoral Symposium, FSE 2014 SEES Symposium, and COMPSAC 2017 SETA Symposium. His research interests include big data software engineering, intelligent software testing and analysis, and adaptive and autonomous software systems. He is a senior member of the IEEE and member of the ACM.

Bingying Guo received her master degree in computer science and technology from Nanjing University in 2018. Her research interests include software engineering and pervasive computing.

Xiaoxing Ma received his doctoral degree in computer science and technology from Nanjing University, China. He is a full professor with the State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology at Nanjing University. His research interests include self-adaptive software systems, cloud computing, software architecture, and middleware systems. He co-authored more than 60 peer-reviewed conference and journal papers, and has served as technical program committee member on various international conferences. He is a member of the IEEE and the ACM.

Jian Lu received his doctoral degree in computer science and technology from Nanjing University, China. He is a full professor with the Department of Computer Science and Technology and 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.