The Journal of Systems and Software 207 (2024) 111852

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Incremental-concurrent fusion checking for efficient context consistency[☆]

Lingyu Zhang, Huiyan Wang[*], Chuyang Chen, Chang Xu, Ping Yu

State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
Department of Computer Science and Technology, Nanjing University, Nanjing, China

ARTICLE INFO

Keywords:
Constraint checking
Context consistency
Check fusion

ABSTRACT

Smart applications can adapt their behaviors based on their understanding to environments (a.k.a. contexts). This capability can, however, incur unexpected misbehavior or even crash, when application contexts are inaccurate or conflicting with each other due to environmental noises. In the recent decade, various constraint checking techniques have been proposed to help validate contexts against consistency constraints, in order to guard context consistency in time. However, with growing environmental dynamics and context volume, it is getting increasingly challenging to ensure context consistency. In this article, we propose a novel approach, INFuse, to fuse together two lines of techniques, namely, incremental checking and concurrent checking, for sound and efficient constraint checking. Realizing such check fusion has to address the challenges rising from the gap between the micro analysis for reusable elements in incremental checking and the macro collection of parallel tasks in concurrent checking. INFuse solves them by automatically deciding maximal concurrent boundaries in a sequence of context changes, and soundly fusing incremental and concurrent checking together for context consistency, with theoretical guarantees. Our experimental evaluation with real-world context data shows that INFuse could improve constraint checking efficiency by 3.0x–120.3x, as compared with existing state-of-the-art techniques, with better checking quality.

1. Introduction

In the software engineering community, consistency management of software artifacts (e.g., edit script (Kehrer et al., 2013), UML models (Bashir et al., 2016; Messaoudi et al., 2017; Wei and Sun, 2021), and XML documents (Nentwich et al., 2002; Reiss, 2006; Handley et al., 2021)) has received much research attention (Brun et al., 2011), and been intensively involved in various software development processes. In the recent decades, there is an increasing demand for managing the consistency of *contexts*, in order to support smart, yet reliable adaptation behaviors in self-adaptive or context-aware applications (Xu et al., 2020). Unlike traditional software artifacts that are typically static or evolve slowly, contexts, representing an application's understanding to its running environment, are typically prone to frequent changes, and thus call for efficient constraint checking techniques for their runtime validation.

Such validation is usually conducted by examining the contexts collected by an application (or its supporting infrastructure) against a set of predefined *consistency constraints* (Nentwich et al., 2002; Wang et al., 2021; Guo et al., 2017). If any constraint violation is detected, it would indicate the occurrence of a *context inconsistency*. Various constraint checking techniques (Nentwich et al., 2002; Wang et al., 2021; Xu et al., 2010, 2013, 2015) have been studied with different efficiency benefits and costs, e.g., xlinkit (Nentwich et al., 2002), working in a full checking way, generating all results as the correctness baseline, PCC (Xu et al., 2010), checking incrementally by reusing previous results for more efficiency, and Con-C (Xu et al., 2013), checking concurrently basic parallel units that carry similar workloads. However, with the increasing growth of environmental dynamics and context volume, it is getting more and more challenging to validate context consistency in a timely manner, thus causing missed inconsistencies or wrong reports (Wang et al., 2021).

One natural intuition is to fuse incremental checking (e.g., PCC (Xu et al., 2010)) and concurrent checking (e.g., Con-C (Xu et al., 2013)) for even higher efficiency. Indeed, they have been developed from two orthogonal research dimensions, but their fusion is actually non-trivial, with no substantial progress after nearly one decade since their initial proposals. The essential challenge probably comes from this gap: incremental checking analyzes in a fine granularity for reusable parts in previous checking results, while concurrent checking requests to maximize parallel tasks. In other words, the former has to accumulate

[☆] Editor: Heiko Koziolek.
[*] Corresponding author at: State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China.
E-mail addresses: zly@smail.nju.edu.cn (L. Zhang), why@nju.edu.cn (H. Wang), chuyangchen2018@outlook.com (C. Chen), changxu@nju.edu.cn (C. Xu), yuping@nju.edu.cn (P. Yu).

https://doi.org/10.1016/j.jss.2023.111852
Received 18 October 2022; Received in revised form 2 February 2023; Accepted 11 September 2023
Available online 18 September 2023

micro parts (since larger parts not easy for analysis), but the latter requires macro arrangements (since smaller parts not effective for concurrency). If one naively injects concurrent checking into incremental checking (e.g., by concurrently conducting the reusable result analysis in a fine granularity), the performance may instead be compromised (e.g., even less efficient than incremental checking, as our later experiments validated). On the other hand, if one aggressively enlarges the analysis granularity of incremental checking, improper grouping of context changes as a whole could instead lead to wrong results, unfortunately denying the purpose of constraint checking itself.

In this article, we propose INFuse (short form for Incremental-CoNcurrent Fusion ChEcking) to address these two challenges from the above gap: (1) *What-correctness* problem: to automatically analyze and decide the boundaries of collected context changes under checking for maximal concurrency (i.e., checking these context changes as a whole guarantees to be correct, as against checking them individually); (2) *How-correctness* problem: to soundly switch between incremental checking and concurrent checking upon the context changes grouped as a whole for higher efficiency (i.e., efficiently conducting both result reusing and parallel analysis). We address both challenges with theoretical guarantees.

We experimentally evaluated INFuse and compared it to existing constraint checking techniques on application scenarios with real-world context data following existing work (Wang et al., 2021; Xu et al., 2010, 2013, 2015). The experimental results show that INFuse could dramatically boost the checking efficiency (up to 120.3x, 62.3x, and 5.7x improvements) by saving checking time (up to 99.2%, 98.4%, and 85.0% time reductions), as compared to existing techniques (ECC, Con-C, and PCC, respectively). When tested in a practical scenario with dynamic changes, INFuse won with extremely high efficiency and almost perfect checking results, while existing techniques suffered down to a 3.3% precision and 1.3% recall, exhibiting INFuse's clear technical superiority and applicability.

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

- We propose a novel constraint checking approach, INFuse, with incremental-concurrent checking techniques soundly fused.
- We prove INFuse's properties, namely, what-correctness for concurrency maximization, and how-correctness for fusion soundness, together contributing to INFuse's checking correctness and high efficiency.
- We studie INFuse's time complexity, formally analyzing its efficiency superiority over existing techniques algorithmically.
- We evaluate INFuse and compared it to state-of-the-art techniques, observing substantial efficiency improvement and desirable checking quality.

We also summarize our major extensions made in this article over the its preliminary conference version (Zhang et al., 2022) below:

- Methodology: We prove two theorems in details about INFuse's what-correctness and how-correctness (Sections 3.2 and 3.3), explain the realization details in applying INFuse in practice (Section 3.4), and analyze INFuse's time complexity and compared it to those of existing checking techniques (Section 3.5);
- Evaluation: We strengthen the scale of experiments (24-h contexts now vs. 3-h contexts originally) for answering three original research questions (RQ1, RQ2, and RQ5 in Section 4), and add two new research questions (RQ3 and RQ4) for studying INFuse fusion mechanism and the impact of complexity factors (Section 4).

The remainder of this article is organized as follows: Section 2 introduces the background and formulates our problem. Section 3 elaborates on our INFuse's methodology with formal complexity analysis. Section 4 evaluates INFuse with real-world application scenarios. Section 5 discusses the related work in recent years, and finally Section 6 concludes this article.

2. Background

2.1. Preliminary

We define a *context* as a piece of information about an application's running environment (e.g., location, user, activity, etc.) (Wang et al., 2021; Xu et al., 2010, 2015). Each context can be modeled as a finite set of relevant elements. For example, in a package delivery application (Wang et al., 2021; Xu et al., 2010) that schedules transportation robots across warehouse, all robots currently in warehouse x can be modeled by a context $C_x = \{r_1, r_2, \dots\}$, in which r_i identifies a specific robot.

We define a *context change* to be an update to an existing context, which can be an *addition change* or *deletion change*. We use symbols ("+", "-") to represent them, respectively. Consider this application with context $C_x = \{r_1, r_2\}$. If robot r_3 enters or r_2 leaves the warehouse, we have context changes $<+, C_x, r_3>$ or $<-, C_x, r_2>$.

We use *context pool* to represent the collection of all contexts interesting to the application. For the aforementioned application, its context pool is $P = \{C_x, C_y\}$, which considers warehouses x and y.

To validate contexts, one could define *consistency constraints* (Nentwich et al., 2002; Wang et al., 2021), which model physical laws or application-specific requirements (Nentwich et al., 2002; Wang et al., 2021; Xu et al., 2010), and check whether any constraint is violated (when yes, an *inconsistency* is detected). Existing work (Wang et al., 2021; Xu et al., 2010, 2015) has mostly followed a first order logic (FOL) styled language to specify consistency constraints:

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

Here, C represents a context; v_i is a variable, taking an element from C as its value; the $bfunc$ terminal is a domain-specific function that takes values of variables as input and returns a Boolean value (True or False). For example, one may define a consistency constraint like "any robot can only be in one warehouse at the same time" (Wang et al., 2021), for the aforementioned application:

$$\mathsf{S}_{\mathsf{loc}} : \forall v_x \in C_x(\texttt{not}(\exists v_y \in C_y(\texttt{Same}(v_x, v_y)))).$$

Incremental checking (Xu et al., 2010) examines each context change to analyze its impact on a constraint's previous checking result, while concurrent checking (Xu et al., 2013) would request multiple context changes for parallelism. In the following, we analyze the challenges when one combines the two techniques together.

2.2. Illustrative example and challenges

Consider our package delivery application with two warehouses (x and y) and three robots (r_1, r_2, and r_3). In this scenario, robot movements are captured by the RFID technology. Suppose that initially robot r_1 is in warehouse x and r_2 in y. However, RFID technology typically suffers from missing reads (Jeffery et al., 2006; Rao et al., 2006; Patil et al., 2015; Fescioglu-Ünver et al., 2015) during this process, and this is common in practical RFID-enhanced sensing. In this scenario, robot r_3 enters warehouse y, and r_2 leaves y and re-enters y. Next, robots r_3 leaves y, enters x, and leaves x in turn. Therefore, we consider such a situation, in which the movement of robot r_3 leaving y is accidentally missed, i.e., $<-, C_y, r_3>$ (chg′) was "*missed*" (five changes remaining), as illustrated in Fig. 1. We call it "missed" here because it is caused by the RFID missing read problem.

When one conducts constraint checking on the context pool *upon each context change* (as the *individual checking* illustrates in Fig. 1) against the aforementioned $\mathsf{S}_{\mathsf{loc}}$ constraint, a context inconsistency inc_1 would be detected at P_4 (suggesting robot r_3 in both warehouses x and y). Incremental checking can work to speed up the checking upon each context change.

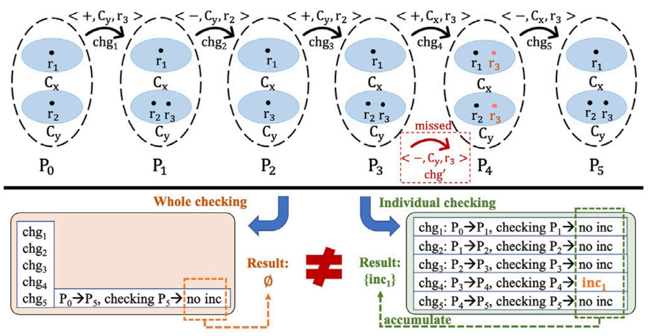
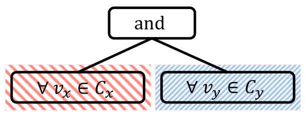
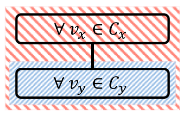


Fig. 1. An illustrative example (P_i is the evolving context pool after each context change).



(a) Parallel structure



(b) Nested structure

Fig. 2. Two structures of consistency constraints.

If one applies concurrent checking, multiple context changes have to be considered for parallelism. Then these changes are applied together and checked *as a whole* (as the *whole checking* illustrates in Fig. 1). However, checking the final context pool P_5 would report no inconsistency. The inconsistency inc_1 is missed (or kept hidden in constraint checking) due to the interference between chg_4 and chg_5. This context inconsistency is missed due to checking certain context changes as a whole, and it is a problem with the constraint checking itself. We explained it by "(the inconsistency) kept hidden in constraint checking", implying that the inconsistency missing is caused by improper grouping of context changes (to be explained later). Therefore, we consider the sequence of these five changes *invalid* for checking together. Then our first question (challenge) arises: *How does one compose constraint checking tasks that both maximize the parallelism (i.e., involving more context changes) and guarantee the validity (i.e., inconsistency never made hidden)?* Fusing incremental checking and concurrent checking together (or *fusion checking*) has to answer this question.

Now suppose that we have obtained a valid constraint checking task, which involves four context changes ($\mathsf{chg}_1, \mathsf{chg}_2, \mathsf{chg}_3, \mathsf{chg}_4$). Then, how can one realize both incremental checking and concurrent checking on these changes? The former handles these changes in turn according to their temporal orders, while the latter parallelizes the handling of these changes without any temporal order. This could induce natural logical conflicts (e.g., considering that change chg_3 is to add an element deleted by chg_2).

To alleviate the complexity, one might consider grouping context changes according to different contexts they relate to, e.g., partitioning context changes into context C_x-related changes and C_y-related changes. Still, checking the two groups concurrently may be intertwined. For a consistency constraint illustrated in Fig. 2(a) with a parallel structure, it could be possible to handle the two groups of context changes concurrently. However, if the constraint has a nested structure as illustrated in Fig. 2(b), the two groups of changes certainly have intertwined impacts on the constraint (i.e., depends-on or subsumed), as concurrent checking would induce unexpected consequences. Therefore, we have the second question (challenge): *How can fusion checking work correctly*?

2.3. Problem formulation

We formulate the preceding two questions (challenges) into two problems, namely, *what-correctness* and *how-correctness*.

Given a sequence of context changes under checking, $(chg_1, chg_2, \ldots)$, P_i represents the evolving context pool after applying change chg_i to existing contexts in pool P_{i-1}. P_i is the collection of all contexts interesting to the concerned application at time t_i (P_0 is the initial pool at time t_0). To be specific, we have used $\mathsf{ideal\_chk}(P_i, s)$ and $\mathsf{chk}(P_i, s)$ to denote the checking functionalities provided by the ideal checking and our fusion checking, respectively, which return reported inconsistencies as the results when examining the contexts in P_i against constraint s. The what-correctness requests that our fusion checking should produce the same checking results by checking context changes as a whole, as compared to checking them individually. That is, it should carefully decide what context changes to check as a whole, so as to avoid any interference inside these changes. Given a checking task $(T = (chg_m, chg_{m+1}, \ldots, chg_n))$, the what-correctness is as follows:

$$\mathsf{chk}(P_n, s) = \bigcup_{i=m}^{n} \mathsf{chk}(P_i, s) \tag{1}$$

The how-correctness requests that our fusion checking should produce the same checking results by fusing incremental and concurrent checking together, as compared to checking directly (e.g., by entire (Nentwich et al., 2002), incremental (Xu et al., 2010), or concurrent checking (Xu et al., 2013)). It is as follows:

$$\mathsf{chk}(P_n, s) = \mathsf{ideal\_chk}(P_n, s) \tag{2}$$

Our fusion checking addresses the two correctness problems in the next section.

3. Methodology

3.1. Approach overview

Fig. 3 overviews our fusion checking (INFuse) approach. It consists of two parts, namely, WHAT-TO-CHECK and HOW-TO-CHECK, targeting at our preceding two challenges, respectively. The first part decides boundaries of context changes that are valid to check as a whole (Section 3.2), and the second part realizes the fusion of incremental and concurrent checking (Section 3.3).

In the first part, INFuse analyzes the impacts of context changes of different types, examines what impacts would cause context inconsistencies hidden, and derives validity criteria for deciding what context changes to group together. In the second part, INFuse checks grouped context changes as a whole using its own incremental-concurrent fusion semantics for inconsistency detection.

3.2. WHAT-TO-CHECK: Task arrangement

INFuse decides proper boundaries in a sequence of context changes, so that each decided group of changes are valid to check as a whole. "Valid" means that no inconsistency would be hidden in the constraint checking. Each valid group of context changes composes a *constraint checking task*.

To decide the validity, we would first investigate the impacts of different context changes on the checking of a given consistency constraint. Specifically, if a context change can cause the constraint's evaluation from True to False, it tends to expose an inconsistency. Otherwise, the change can cause the constraint's evaluation from False to True, and it tends to hide an inconsistency. The insight of INFuse is to analyze and avoid the combination of such two context changes (otherwise, the first inconsistency might thus become hidden), but the challenge is that INFuse has to decide it before actual evaluation. Later, based on such impact analysis, INFuse derives validity criteria

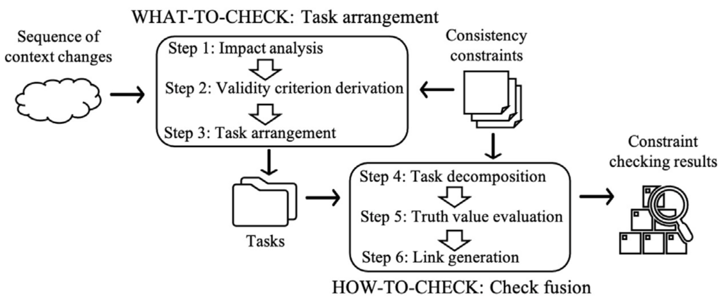

WHAT-TO-CHECK: Task arrangement
Sequence of context changes
Step 1: Impact analysis
Step 2: Validity criterion derivation
Step 3: Task arrangement
Consistency constraints
Tasks
Step 4: Task decomposition
Step 5: Truth value evaluation
Step 6: Link generation
HOW-TO-CHECK: Check fusion
Constraint checking results

Fig. 3. Overview of our INFuse approach.

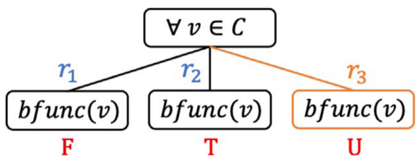


Fig. 4. Example of a universal formula.

Table 1
Base impact.

Context change	$\forall v \in C(f)$	$\exists v \in C(f)$
<+ , C, U>	{m_{TT}, m_{TF}, m_{FF}}	{m_{TT}, m_{FT}, m_{FF}}
<-, C, T>	{m_{TT}, m_{FF}}	{m_{TT}, m_{TF}}
<-, C, F>	{m_{FT}, m_{FF}}	{m_{TT}, m_{FF}}
<-, C, U>	{m_{TT}, m_{FT}, m_{FF}}	{m_{TT}, m_{TF}, m_{FF}}

for constraint checking tasks, and arranges context changes into proper groups.

We elaborate on our idea in three steps.

Step 1: Impact analysis. We now model more precisely a context change in a form of < *type*, *context*, *truthvalue* >. A truth value has only two values, i.e., True and False. When we talk about the truth value of a consistency constraint, it must be one of them. Nevertheless, when we model the impact of a context change to the evaluation result of a constraint, we have to distinguish two cases (already knowing the truth value of a certain formula vs. not knowing yet). Such treatment appears only during the impact analysis, and will not affect final truth values. Thus, *truthvalue* here is either a specific truth value (T or F) or a unknown truth value (U).

Then all context changes can be partitioned into four cases: <+, C, U>, < −, C, T>, < −, C, F>, and < −, C, U>. Here, <+, C, U> denotes an addition change to context C, with its associated formula not evaluated yet (U: Unevaluated); < −, C, T> denotes a deletion change to context C, with its associated formula previously evaluated to True (T: True; F: False). For example, consider constraint $\forall v \in C(bfunc(v))$ and context $C = \{r_1, r_2\}$ as illustrated in Fig. 4 (truth values annotated). The impact of any addition change (e.g., <+, C, r_3>) can be represented by <+, C, U> since the newly element r_3 has not been evaluated yet for $bfunc$. The impact of a deletion change has three cases according to the previous truth value of the element to delete for $bfunc$: (1) < −, C, T>, if the element to delete has been evaluated to True, e.g., < −, C, r_2>; (2) < −, C, F>, if the element has been evaluated to False, e.g., < −, C, r_1>; (3) < −, C, U>, when the element is just added and has not been evaluated yet, e.g., < −, C, r_3>.

We note that only universal and existential formulas are associated with contexts in consistency constraints, and thus context changes directly affect such formulas (named *base formulas*). Consider our preceding constraint S_{loc} (Section 2.1). Change < −, C_y, r_2> directly affects the constraint's existential quantifier part ($\exists v_y \in C_y$) and makes formula $\exists v_y \in C_y(\mathtt{Same}(v_x, v_y))$ its base formula. In our illustrative example in Fig. 1, chg_1, chg_3 and chg_4 are three addition changes and all belong to the impact case <+, C_x, U> or <+, C_y, U>. Suppose that the constraint has been evaluated on P_0. Then chg_2 belongs to the case of < −, C_y, F> and chg_5 belongs to < −, C_x, U>.

Next we analyze how a context change produces its impact (a.k.a. *base impact*) to the concerned base formula, and then track the impact to the whole constraint (a.k.a. *overall impact*) containing this formula.

The base impact has four kinds, namely, m_{TT}, m_{TF}, m_{FT}, and m_{FF}, representing the truth value of a formula keeping True, changing from True to False, from False to True, and keeping False, respectively. Table 1 lists all base impacts that can be produced by each particular context change to each possible base formula. Take the universal formula $\forall v \in C(f)$ as an example. Change <+, C, U> can produce all impacts except m_{FT}, because adding an element into a context can never make the universal formula evaluated from False to True, while < −, C, T> can produce only m_{TT} and m_{FF}, because deleting an element from a context with truth value of True can never make the universal formula evaluated from True to False or from False to True. Other cases can be explained similarly.

Then we follow the tracking rules in Fig. 5 to decide how the overall impact of a particular context change on a consistency constraint depends on the base impact of this change on its associated base formula.

Take universal formula $g := \forall v \in C(f)$ for example. We consider all four impacts: (1) if a change has impact m_{TT} on f, it leads to g remaining its previous truth value, i.e., having impact m_{TT} or m_{FF}; (2) if the change has impact m_{TF}, it can cause g evaluated to False, i.e., having impact m_{TF} or m_{FF}; (3) if the change has impact m_{FF}, it makes g keep evaluated to False, i.e., having impact m_{FF}; (4) if the change has impact m_{FT}, it can cause g to keep evaluated to False or from False to True, i.e., having impact m_{FF} or m_{FT}. Combining all cases together, the impact on the universal formula g should be $\mathrm{impact}(f) \cup \{m_{FF}\}$. Recursively, one can continue to track the impact down to formula f. If the tracking already reaches the base formula the specific change concerns, then the tracking can terminate with the associated base impact. Other tracking rules can be explained similarly.

For example, consider context change chg_1 =<+, C_y, r_3> in Fig. 1. We model it by <+, C_y, U>, and analyze its overall impact on constraint S_{loc} as follows:

$$
\begin{aligned}
&\mathrm{impact}(chg_1, \forall v_x \in C_x(\mathtt{not}(\exists v_y \in C_y(\mathtt{Same}(v_x, v_y)))))\\
&\quad = \mathrm{impact}(chg_1, \mathtt{not}(\exists v_y \in C_y(\mathtt{Same}(v_x, v_y)))) \cup \{m_{FF}\}\\
&\quad = \mathrm{flipSet}(\mathrm{impact}(chg_1, \exists v_y \in C_y(\mathtt{Same}(v_x, v_y)))) \cup \{m_{FF}\}\\
&\quad = \mathrm{flipSet}(\mathrm{base\_impact}(chg_1, \exists)) \cup \{m_{FF}\}\\
&\quad = \mathrm{flipSet}(\{m_{TT}, m_{FT}, m_{FF}\}) \cup \{m_{FF}\}\\
&\quad = \{m_{FF}, m_{TF}, m_{TT}\}
\end{aligned}
$$

After analyzing the overall impact of a context change, we dynamically update the evaluation situation of the formulas directly or indirectly affected by this context change, in order to model its next

Auxiliary functions :

– impact, where $\mathsf{impact}(chg, f)$ refers to chg's impact on f.

– base_impact, where $\mathsf{base\_impact}(chg, \exists/\forall)$ follows Table 1.

– flip, where $\mathsf{flip}(\mathsf{m_{TT}}) := \mathsf{m_{FF}}$; $\mathsf{flip}(\mathsf{m_{FF}}) := \mathsf{m_{TT}}$; $\mathsf{flip}(\mathsf{m_{TF}}) := \mathsf{m_{FT}}$;

$\mathsf{flip}(\mathsf{m_{FT}}) := \mathsf{m_{TF}}$;

– flipSet, where $\mathsf{flipSet}(M) := \{\mathsf{flip}(m) \mid m \in M\}$.

Tracking rules :

– $\mathsf{impact}(chg, \forall v \in C(f)) =$

(1) $\mathsf{base\_impact}(chg, \forall)$, when chg affects C,

(2) $\mathsf{impact}(chg, f) \cup \{\mathsf{m_{FF}}\}$, when chg affects f;

– $\mathsf{impact}(chg, \exists v \in C(f)) =$

(1) $\mathsf{base\_impact}(chg, \exists)$, when chg affects C,

(2) $\mathsf{impact}(chg, f) \cup \{\mathsf{m_{TT}}\}$, when chg affects f;

– $\mathsf{impact}(chg, \text{not } (f)) = \mathsf{flipSet}(\mathsf{impact}(chg, f))$;

– $\mathsf{impact}(chg, (f_1) \text{ and } (f_2)) =$

(1) $\mathsf{impact}(chg, f_1) \cup \{\mathsf{m_{FF}}\}$, when chg affects f_1,

(2) $\mathsf{impact}(chg, f_2) \cup \{\mathsf{m_{FF}}\}$, when chg affects f_2;

– $\mathsf{impact}(chg, (f_1) \text{ or } (f_2)) =$

(1) $\mathsf{impact}(chg, f_1) \cup \{\mathsf{m_{TT}}\}$, when chg affects f_1,

(2) $\mathsf{impact}(chg, f_2) \cup \{\mathsf{m_{TT}}\}$, when chg affects f_2;

– $\mathsf{impact}(chg, (f_1) \text{ implies } (f_2)) =$

(1) $\mathsf{flipSet}(\mathsf{impact}(chg, f_1)) \cup \{\mathsf{m_{TT}}\}$, when chg affects f_1,

(2) $\mathsf{impact}(chg, f_2) \cup \{\mathsf{m_{TT}}\}$, when chg affects f_2.

Fig. 5. Tracking rules.

context change precisely. For example, consider the context pool P_0 in Fig. 1, the universal formula associated with $\mathsf{r_1}$ and the existential formula associated with $\mathsf{r_2}$ are both evaluated as True because there is no inconsistency. After analyzing context change $\mathsf{chg_1}$ = <+, C_y, $\mathsf{r_3}$>, the existential formula associated with $\mathsf{r_3}$ is unevaluated, changing the evaluation of universal formula associated with $\mathsf{r_1}$ from True to Unevaluated. In this way, the overall impacts of changes $\mathsf{chg_2}$, $\mathsf{chg_3}$, $\mathsf{chg_4}$, and $\mathsf{chg_5}$ in Fig. 1 can be obtained similarly, i.e., $\{\mathsf{m_{TT}}, \mathsf{m_{FF}}\}$, $\{\mathsf{m_{TT}}, \mathsf{m_{TF}}, \mathsf{m_{FF}}\}$, $\{\mathsf{m_{TT}}, \mathsf{m_{TF}}, \mathsf{m_{FF}}\}$, and $\{\mathsf{m_{TT}}, \mathsf{m_{FT}}, \mathsf{m_{FF}}\}$.

Step 2: Validity criterion derivation. With analyzed impacts of context changes, we proceed to classify them into three categories according to how they affect the detection of context inconsistencies.

Definition 1 (*Inc-Exposing Change*)**.** Given a consistency constraint s, if the overall impact of a context change contains $\mathsf{m_{TF}}$ but no $\mathsf{m_{FT}}$, it is an inc-exposing change (or E-change), suggesting possibly exposing a new inconsistency for s.

Definition 2 (*Inc-Hiding Change*)**.** Given a constraint s, if the overall impact of a change contains $\mathsf{m_{FT}}$ but no $\mathsf{m_{TF}}$, it is an inc-hiding change (or H-change), suggesting possibly hiding an existing inconsistency for s.

Definition 3 (*Inc-Irrelevant Change*)**.** Given a constraint s, if the overall impact of a change contains neither $\mathsf{m_{FT}}$ nor $\mathsf{m_{TF}}$, it is an inc-irrelevant change (or I-change), suggesting irrelevant to detecting any inconsistency.

Note that no context change has both types $\mathsf{m_{FT}}$ and $\mathsf{m_{TF}}$, since (1) any base impact contains at most one such type (Table 1), and (2) tracking rules never breaks this property (Fig. 5). Therefore, E-change, H-change, and I-change are *complete*.

Based on the above definitions, if a constraint checking task contains any ordered E-change (with $\mathsf{m_{TF}}$) and H-change (with $\mathsf{m_{FT}}$) pair in its sequence of context changes, it is invalid to check these changes as a

Algorithm 1: Task arrangement

```
Input  : set of consistency constraints $S$, new context change $chg_{new}$
Output: set of consistency constraints $S$ (updated)
for each $s \in S$ do
    $p$ = impact ($chg_{new}$, $s$);
    if $p$ contains $m_{FT}$ then
        $chg_{new}.type$ = H-change;
    else if $p$ contains $m_{TF}$ then
        $chg_{new}.type$ = E-change;
    else
        $chg_{new}.type$ = I-change;
    if $chg_{new}.type$ == H-change then
        for each change $chg$ in $s.Task$ do
            if $chg.type$ == E-change then
                fusionchecking($s.Task$, $s$);
                $s.Task$.clear();
                break;
    $s.Task$ ← append($chg_{new}$);
return $S$;
```

whole (i.e., inconsistency possibly hidden). Based on this observation, we derive our validity criterion as follows:

Definition 4 (*Validity Criterion*)**.** Given a constraint checking task with a sequence of context changes, if the sequence contains any ordered E-change and H-change pair (either contiguous or not), it is an invalid task; otherwise, valid.

Consider our preceding illustrative example in Fig. 1. Context changes chg_1 (<+, C_y, U>), chg_3 (<+, C_y, U>), and chg_4 (<+, C_x, U>) all have the m_{TF} impact (i.e., E-change), change chg_5 (< −, C_y, U>) has the m_{FT} impact (i.e., H-change), and the remaining change chg_2 has neither of them (i.e., I-change).

Then, consider two tasks: $T_1 = (\text{chg}_1, \text{chg}_2, \text{chg}_3, \text{chg}_4, \text{chg}_5)$, and $T_2 = (\text{chg}_1, \text{chg}_2, \text{chg}_3, \text{chg}_4)$. T_1 contains an E-change and H-change (chg_5) pair, thus invalid. T_2 does not contain any such pair, thus valid. The results match our earlier analysis in Section 2.2.

Step 3: Task arrangement. With the above validity criterion, INFuse can compose constraint checking tasks with valid context changes only.

Algorithm 1 explains how to arrange valid constraint checking tasks. Given a consistency constraint s, when context change chg_{new} is collected, INFuse first analyzes its impact on s to decide its category (Lines 2–8), i.e., E-/H-/I-change. Then, if chg_{new} is an H-change, INFuse examines whether there is any existing E-change chg in the current task. If yes (Line 11), INFuse conducts fusion checking with all existing changes in the task (details to be discussed later in the HOW-TO-CHECK part) (Line 12), and finishes this task (s's new task starts with chg_{new}, Lines 13–14). Otherwise, INFuse keeps maximizing a constraint checking task until any possible E-change and H-change pair occurs.

We give the following theorem to guarantee that INFuse always returns the same checking result by its whole checking of thus arranged tasks, as compared to individual checking.

Theorem 1 (***WHAT-Correctness***)**.** *Given any consistency constraint and associated context pool,* INFuse *produces the same result for its arranged valid context changes, no matter it checks these changes as a whole or individually.*

Proof. Let the concerned constraint be s with the associated context pool P_0. INFuse's arranged valid context changes compose a constraint checking task $T = (chg_1, \cdots, chg_n)$. P_i represents the context pool right after applying context change chg_i. As discussed in Section 2.3, in order to prove this WHAT-Correctness theorem, we actually aim to prove:

$$\text{chk}(P_n, s) = \bigcup_{i=1}^{n} \text{chk}(P_i, s) \tag{3}$$

To get Eq. (3), one can prove that checking results for $P_0, \ldots, P_{n-1}$ are all subsets of the checking result for the checking result for P_n. This target (i.e., the following Eq. (4)) serves as a sufficient condition for Eq. (3), i.e.,

$$\bigcup_{i=1}^{n-1} \text{chk}(P_i, s) \subseteq \text{chk}(P_n, s) \tag{4}$$

We use reduction to absurdity by assuming that Eq. (4) does not hold. That is, there is an inc_x satisfying:

$$inc_x \in (\bigcup_{i=1}^{n-1} \text{chk}(P_i, s)) \setminus \text{chk}(P_n, s) \tag{5}$$

Suppose inc_x is first exposed by chg_j $(1 \le j < n)$, i.e., $inc_x \in \text{chk}(P_j, s)$ and $inc_x \notin \text{chk}(P_{j-1}, s)$. Due to our definition of E/H/I-changes, chg_j is an E-change. Moreover, since $inc_x \notin \text{chk}(P_n, s)$, it should be hidden no later than chg_n is applied and checked. Suppose inc_x is actually hidden by chg_k $(j < k \le n)$, i.e., $inc_x \notin \text{chk}(P_k, s)$. By definition, chg_k must be an H-change. Therefore, we can derive that:

$$inc_x \in \text{chk}(P_j, s), \tag{6}$$

$$inc_x \notin \text{chk}(P_k, s). \tag{7}$$

This actually denotes that inc_x was first exposed by an E-change chg_j, and then hidden by a H-change chg_k, which clearly violates the nonexistence of an ordered E-change and H-change in any constraint checking task according to the validity criterion (Definition 4). Therefore, this leads to a contradiction to our assumption, so Eq. (4) holds and thus Eq. (3) can be easily proved as such. This completes our proof. □

In the following, we explain how INFuse fuses incremental and concurrent checking to efficiently and soundly handle valid context changes in each task.

3.3. HOW-TO-CHECK: Check fusion

Given a valid constraint checking task, INFuse fuses incremental and concurrent checking and treats all context changes in the task as a whole for efficiency. INFuse first decomposes all changes in a task into several subsets based on their nature, and then conducts constraint checking by two steps, namely, truth value evaluation and link generation, which examines whether the concerned consistency constraint is violated and why the violation, if any, occurs.

Step 4: Task decomposition. INFuse first decomposes all context changes (addition or deletion) in the given constraint checking task into three subsets, namely, *truly added* set (or $ASet$ for short), *truly deleted* set ($DSet$) and *updated set* ($USet$) for each consistency constraint. They contain *truly added* elements (i.e., not deleted later), *truly deleted* elements (not added back later) and *updated* elements (i.e., deleted first and added back), respectively. Suppose that context C eventually becomes C' after applying all relevant changes in task T. Then the three sets can be calculated: $ASet = C' \setminus C$, $DSet = C \setminus C'$, and $USet = \{e | e \in C \cap C' \wedge \exists\ chg \in T(chg = \langle +/-, C, e \rangle)\}$.

We define the Affected function to indicate whether a formula itself or its subformula is affected by the context changes in a constraint checking task. Given a formula from a consistency constraint, the Affected function returns T (means True) if and only if the formula itself or its subformula references a context involved in the $ASet$, $DSet$ or $USet$ associated with this constraint; otherwise, F (means False).

$$\tau_{\mathsf{partial}}[\forall v \in C(f)]_\alpha =$$

$$(1)\ \tau_0[\forall v \in C(f)]_\alpha,\ \text{if Affected}(f) = \mathsf{F} \text{ and } (ASet = \emptyset \text{ and } DSet = \emptyset \text{ and } USet = \emptyset).$$

$$(2)\ \tau_0[\forall v \in C(f)]_\alpha \wedge t_1 \wedge \cdots \wedge t_a, \text{where } (t_1, \cdots, t_a) = \mathsf{eval}_{\mathsf{entire}}(\tau[f]_{\mathsf{bind}((v,y_j),\alpha)} \mid y_j \in ASet),$$
$$\text{if Affected}(f) = \mathsf{F} \text{ and } (ASet \neq \emptyset \text{ and } DSet = \emptyset \text{ and } USet = \emptyset).$$

$$(3)\ \mathsf{T} \wedge \tau_0[f]_{\mathsf{bind}((v,x_1),\alpha)} \wedge \cdots \wedge \tau_0[f]_{\mathsf{bind}((v,x_{n-a-u}),\alpha)} \wedge t_1 \wedge \cdots \wedge t_{a+u} \mid x_i \in C \setminus (ASet \cup USet)),$$
$$\text{where } (t_1, \cdots, t_{a+u}) = \mathsf{eval}_{\mathsf{entire}}(\tau[f]_{\mathsf{bind}((v,y_j),\alpha)} \mid y_j \in ASet \cup USet),$$
$$\text{if Affected}(f) = \mathsf{F} \text{ and } (DSet \neq \emptyset \text{ or } USet \neq \emptyset).$$

$$(4)\ \mathsf{T} \wedge t_1 \wedge \cdots \wedge t_n, \text{where } (t_1, \cdots, t_n) = \mathsf{eval}_{\mathsf{partial}}(\tau[f]_{\mathsf{bind}((v,x_i),\alpha)} \mid x_i \in C),$$
$$\text{if Affected}(f) = \mathsf{T} \text{ and } (ASet = \emptyset \text{ and } DSet = \emptyset \text{ and } USet = \emptyset).$$

$$(5)\ \mathsf{T} \wedge t_1 \wedge \cdots \wedge t_n, \text{where } (t_1, \cdots, t_{a+u}) = \mathsf{eval}_{\mathsf{entire}}(\tau[f]_{\mathsf{bind}((v,y_j),\alpha)} \mid y_j \in ASet \cup USet)$$
$$\text{and } (t_{a+u+1}, \cdots, t_n) = \mathsf{eval}_{\mathsf{partial}}(\tau[f]_{\mathsf{bind}((v,x_i),\alpha)} \mid x_i \in C \setminus (ASet \cup USet)),$$
$$\text{if Affected}(f) = \mathsf{T} \text{ and } (ASet \neq \emptyset \text{ or } DSet \neq \emptyset \text{ or } USet \neq \emptyset).$$

Fig. 6. INFUSE's partial truth value evaluation semantics for the universal formula.

$$\tau_{\mathsf{entire}}[\forall v \in C(f)]_\alpha = \mathsf{T} \wedge \tau_{\mathsf{entire}}[f]_{\mathsf{bind}((v,x_1),\alpha)} \wedge \cdots \wedge \tau_{\mathsf{entire}}[f]_{\mathsf{bind}((v,x_n),\alpha)} | x_i \in C$$

Fig. 7. INFUSE's entire truth value evaluation semantics for the universal formula.

INFUSE would rely on the three subsets to decide when to switch between incremental checking (by partial checking semantics later) and concurrent checking (by entire checking semantics later). The checking is composed of the truth value evaluation (returning T or F) and link generation (returning links (Xu et al., 2010)). The following gives an example link for our preceding inconsistency detected in the illustrative example (more explanation about link is given later in Step 6): $(\mathsf{violated}, \{(v_\mathsf{x} = \mathsf{r}_3), (v_\mathsf{y} = \mathsf{r}_3)\})$.

Step 5: Truth value evaluation. We use $\tau_{\text{INFUSE}}[s]$ to represent INFUSE's truth value evaluation on consistency constraint s. τ_{INFUSE} starts with incremental checking by invoking its partial checking semantics, i.e., $\tau_{\text{INFUSE}}[s] = \tau_{\mathsf{partial}}[s]_\alpha$. Here, α is the variable assignment, which is empty at the beginning and updated later by the bind function when evaluating universal or existential subformula in constraint s to add new variable bindings into α. In the following, we take the universal formula as an example to explain INFUSE's truth value evaluation. A full treatment of all formula types is accessible at our Appendix.

Consider universal formula $\forall v \in C(f)$. Suppose that all context changes in a constraint checking task have been decomposed into related $ASet$, $DSet$, and $USet$. Fig. 6 gives INFUSE's partial truth value evaluation semantics (five cases).

(1) If no change affects the universal formula or its subformula, then this formula's previous truth value τ_0 is reusable.
(2) If the changes affect the universal formula only by adding new elements into context C only, then this formula's previous truth value τ_0 is reusable, and one can update it with evaluation results of the new elements from $ASet$, by the $\mathsf{eval}_{\mathsf{entire}}$ function in Fig. 8 and τ_{entire} semantics in Fig. 7 ("entire" due to new elements (no reusable results); concurrent evaluations may be applied (explained later)).
(3) If the changes affect the universal formula only by deleting existing elements from, or updating them in, context C, then the evaluation results of the remaining elements in C (i.e., $C \setminus (ASet \cup USet)$) are reusable, and those of the other elements should be calculated by the $\mathsf{eval}_{\mathsf{entire}}$ function similarly.
(4) If the changes affect the subformula only, then the evaluation results of all elements in C should be updated by the $\mathsf{eval}_{\mathsf{partial}}$ function in Fig. 8 ("partial" due to elements not changed (some reusability possible)).

$$\mathsf{eval}_{\mathsf{entire}}(\tau[f]_{\mathsf{bind}((v,x_i),\alpha)} \mid x_i \in Set) =$$
$$(1)\ \tau_{\mathsf{entire}}[f]_{\mathsf{bind}((v,x_1),\alpha)} \parallel \cdots \parallel \tau_{\mathsf{entire}}[f]_{\mathsf{bind}((v,x_s),\alpha)},$$
$$\text{if } \forall v \in C(f) \text{ is a concurrent point;}$$
$$(2)\ \tau_{\mathsf{entire}}[f]_{\mathsf{bind}((v,x_1),\alpha)}\ ;\ \cdots\ ;\ \tau_{\mathsf{entire}}[f]_{\mathsf{bind}((v,x_s),\alpha)},$$
$$\text{otherwise.}$$

$$\mathsf{eval}_{\mathsf{partial}}(\tau[f]_{\mathsf{bind}((v,x_i),\alpha)} \mid x_i \in Set) =$$
$$(1)\ \tau_{\mathsf{partial}}[f]_{\mathsf{bind}((v,x_1),\alpha)} \parallel \cdots \parallel \tau_{\mathsf{partial}}[f]_{\mathsf{bind}((v,x_s),\alpha)},$$
$$\text{if } \forall v \in C(f) \text{ is a concurrent point;}$$
$$(2)\ \tau_{\mathsf{partial}}[f]_{\mathsf{bind}((v,x_1),\alpha)}\ ;\ \cdots\ ;\ \tau_{\mathsf{partial}}[f]_{\mathsf{bind}((v,x_s),\alpha)},$$
$$\text{otherwise.}$$

Fig. 8. Semantics of the eval functions (entire and partial checking).

(5) Otherwise, the changes affect both the universal formula and its subformula, then one has to update the evaluation results of unchanged elements (i.e., $C \setminus (ASet \cup USet)$) by the $\mathsf{eval}_{\mathsf{partial}}$ function and those of changed elements ($(ASet \cup USet)$) by the $\mathsf{eval}_{\mathsf{entire}}$ function.

We note that in the $\mathsf{eval}_{\mathsf{entire}}$ and the $\mathsf{eval}_{\mathsf{partial}}$ functions, concurrent checking can be applied to conduct parallel evaluations as in Fig. 8 ("$\parallel$" means concurrent and ";" means sequential), since these evaluations are independent of each other.

Concurrent points are the places where concurrent checking starts with multi-threading support. As illustrated in Fig. 8 and later Fig. 11, concurrent points are associated with universal or existential formulas, as their subformulas would incur similar checking workloads. Consider our preceding consistency constraint $\mathsf{S}_{\mathsf{loc}}$ and a checking task $T = (\mathsf{chg}_1, \mathsf{chg}_2, \mathsf{chg}_3, \mathsf{chg}_4)$. These changes affect both the constraint's universal formula (i.e., $\forall v_x \in C_x$) and its inner existential formula (i.e., $\exists v_y \in C_y$) in $\mathsf{S}_{\mathsf{loc}}$. They are both concurrent point candidates for starting concurrent checking. We will discuss how to decide proper concurrent points later in Section 3.4.

$$
\begin{aligned}
&\mathcal{L}_{\text{partial}}[\forall v \in C(f)]_\alpha = \\
&\quad (1)\ \mathcal{L}_0[\forall v \in C(f)]_\alpha,\ \text{if Affected}(f) = \mathsf{F} \text{ and } (ASet = \emptyset \text{ and } DSet = \emptyset \text{ and } USet = \emptyset). \\
&\quad (2)\ \mathcal{L}_0[\forall v \in C(f)]_\alpha \cup (\{(\text{violated}, \{v, y_1\})\} \otimes l_1) \cup \cdots \cup (\{(\text{violated}, \{v, y_{a'}\})\} \otimes l_{a'}), \\
&\qquad \text{where } (l_1, \cdots, l_{a'}) = \text{gen}_{\text{entire}}(\mathcal{L}[f]_{\text{bind}((v,y_j),\alpha)} \mid y_j \in ASet \wedge \tau[f]_{\text{bind}((v,y_j),\alpha)} = \mathsf{F}), \\
&\quad \text{if Affected}(f) = \mathsf{F} \text{ and } (ASet \neq \emptyset \text{ and } DSet = \emptyset \text{ and } USet = \emptyset). \\
&\quad (3)\ (\{(\text{violated}, \{v, y_1\})\} \otimes l_1) \cup \cdots \cup (\{(\text{violated}, \{v, y_{a'+u'}\})\} \otimes l_{a'+u'}) \cup \\
&\qquad \{l \mid l \in \{(\text{violated}, \{(v, x_i)\})\} \otimes \mathcal{L}_0[f]_{\text{bind}((v,x_i),\alpha)}\} \mid x_i \in C \setminus (ASet \cup USet) \wedge \tau[f]_{\text{bind}((v,x_i),\alpha)} = \mathsf{F}, \\
&\qquad \text{where } (l_1, \cdots, l_{a'+u'}) = \text{gen}_{\text{entire}}(\mathcal{L}[f]_{\text{bind}((v,y_j),\alpha)} \mid y_j \in ASet \cup USet \wedge \tau[f]_{\text{bind}((v,y_j),\alpha)} = \mathsf{F}), \\
&\quad \text{if Affected}(f) = \mathsf{F} \text{ and } (DSet \neq \emptyset \text{ or } USet \neq \emptyset). \\
&\quad (4)\ \emptyset \cup (\{(\text{violated}, \{v, x_1\})\} \otimes l_1) \cup \cdots \cup (\{(\text{violated}, \{v, x_{n'}\})\} \otimes l_{n'}), \\
&\qquad \text{where } (l_1, \cdots, l_{n'}) = \text{gen}_{\text{partial}}(\mathcal{L}[f]_{\text{bind}((v,x_i),\alpha)} \mid x_i \in C \wedge \tau[f]_{\text{bind}((v,x_i),\alpha)} = \mathsf{F}), \\
&\quad \text{if Affected}(f) = \mathsf{T} \text{ and } (ASet = \emptyset \text{ and } DSet = \emptyset \text{ and } USet = \emptyset). \\
&\quad (5)\ \emptyset \cup (\{(\text{violated}, \{v, y_1\})\} \otimes l_1) \cup \cdots \cup (\{(\text{violated}, \{v, y_{n'}\})\} \otimes l_{n'}), \\
&\qquad \text{where } (l_1, \cdots, l_{a'+u'}) = \text{gen}_{\text{entire}}(\mathcal{L}[f]_{\text{bind}((v,y_j),\alpha)} \mid y_j \in ASet \cup USet \wedge \tau[f]_{\text{bind}((v,y_j),\alpha)} = \mathsf{F}) \\
&\qquad \text{and } (l_{a'+u'+1}, \cdots l_{n'}) = \text{gen}_{\text{partial}}(\mathcal{L}[f]_{\text{bind}((v,x_i),\alpha)} \mid x_i \in C \setminus (ASet \cup USet) \wedge \tau[f]_{\text{bind}((v,x_i),\alpha)} = \mathsf{F}), \\
&\quad \text{if Affected}(f) = \mathsf{T} \text{ and } (ASet \neq \emptyset \text{ or } DSet \neq \emptyset \text{ or } USet \neq \emptyset).
\end{aligned}
$$

Fig. 9. INFuse's partial link generation semantics for the universal formula.

$$
\begin{aligned}
&\mathcal{L}_{\text{entire}}[\forall v \in C(f)]_\alpha = \\
&\quad \{l \mid l \in \{(\text{violated}, \{(v, x_i)\})\} \otimes \mathcal{L}_{\text{entire}}[f]_{\text{bind}((v,x_i),\alpha)}\} \mid x_i \in C \wedge \tau[f]_{\text{bind}((v,x_i),\alpha)} = \mathsf{F}).
\end{aligned}
$$

Fig. 10. INFuse's entire link generation semantics for the universal formula.

$$
\begin{aligned}
&\text{gen}_{\text{entire}}(\mathcal{L}[f]_{\text{bind}((v,x_i),\alpha)} | x_i \in Set \wedge \tau[f]_{\text{bind}((v,x_i),\alpha)} = \mathsf{F}) \\
&\quad (1)\ \mathcal{L}_{\text{entire}}[f]_{\text{bind}((v,x_1),\alpha)} \parallel \cdots \parallel \mathcal{L}_{\text{entire}}[f]_{\text{entire}((v,x_s),\alpha)}, \\
&\qquad \text{if } \forall v \in C(f) \text{ is a concurrent point.} \\
&\quad (2)\ \mathcal{L}_{\text{entire}}[f]_{\text{bind}((v,x_1),\alpha)}\ ;\ \cdots\ ;\ \mathcal{L}_{\text{entire}}[f]_{\text{bind}((v,x_s),\alpha)}, \\
&\qquad \text{otherwise.} \\
&\text{gen}_{\text{partial}}(\mathcal{L}[f]_{\text{bind}((v,x_i),\alpha)} \mid x_i \in Set \wedge \tau[f]_{\text{bind}((v,x_i),\alpha)} = \mathsf{F}) \\
&\quad (1)\ \mathcal{L}_{\text{partial}}[f]_{\text{bind}((v,x_1),\alpha)} \parallel \cdots \parallel \mathcal{L}_{\text{partial}}[f]_{\text{bind}((v,x_s),\alpha)}, \\
&\qquad \text{if } \forall v \in C(f) \text{ is a concurrent point.} \\
&\quad (2)\ \mathcal{L}_{\text{partial}}[f]_{\text{bind}((v,x_1),\alpha)}\ ;\ \cdots\ ;\ \mathcal{L}_{\text{partial}}[f]_{\text{bind}((v,x_s),\alpha)}, \\
&\qquad \text{otherwise.}
\end{aligned}
$$

Fig. 11. Semantics of the gen functions (entire and partial checking).

Step 6: Link generation. Similarly, link generation $\mathcal{L}_{\text{INFuse}}[s]$ in INFuse starts with incremental checking by invoking its partial checking semantics, i.e., $\mathcal{L}_{\text{INFuse}}[s] = \mathcal{L}_{\text{partial}}[s]_\alpha$.

Links are generated to explain why a consistency constraint has been violated or satisfied, in a form of (linkType, variable assignments). The linkType is violated or satisfied, corresponding to the evaluated truth value of False or True, and variable assignments disclose that the violation or satisfaction occurs under what kind of variable bindings. Recalling our preceding example of link (violated, $\{(v_x, r_3), (v_y, r_3)\}$), it means that the preceding constraint S_{loc} is violated when variable v_x and variable v_y are both assigned with r_3. Similarly, Fig. 9 gives INFuse's partial link generation semantics for the universal formula (five cases simiplified; a full treatment of all formula types is accessible at our Appendix).

(1) If no change affects the universal formula or its subformula, this formula's previous link result $\mathcal{L}_0$ is reusable.
(2) If the changes affect the universal formula only by adding new elements, this formula's previous link result $\mathcal{L}_0$ is reusable and one can update it with the link results of the new elements, by the $\text{gen}_{\text{entire}}$ function in Fig. 11 and $\mathcal{L}_{\text{entire}}$ semantics in Fig. 10. Here, the $\otimes$ operator concatenates the base link set of the universal formula (i.e., $\{(\text{violated}, \{v, y_j\})\}$ and link set generated by the subformula (i.e., l_j) by applying a *Concatenate* function to the link pairs formed by link (violated, $\{v, y_j\}$) and every link from l_j. The Concatenate function combines the two links with the same linkType into a new link, which consists of this linkType and the union of all concerned variable assignments from the two links. Their formal definitions can be found in Appendix.
(3) If the changes affect the universal formula only by deleting or updating existing elements, the link results of the remaining elements are reusable, and those of the other elements should be calculated by the $\text{gen}_{\text{entire}}$ function similarly.
(4) If the changes affect the subformula only, the link results of all elements should be updated by the $\text{gen}_{\text{partial}}$ function in Fig. 11.
(5) Otherwise, the changes affect both the universal formula and its subformula, one has to update the link results of unchanged elements by the $\text{gen}_{\text{partial}}$ function and those of changed elements by the $\text{gen}_{\text{entire}}$ function.

Similarly, the $\text{gen}_{\text{entire}}$ and $\text{gen}_{\text{partial}}$ functions can work concurrently for efficiency at concurrent points. In the following, we give the second theorem to guarantee that INFuse soundly fuses incremental and concurrent checking semantics.

Theorem 2 (***HOW-Correctness***)**.** *Given any consistency constraint and associated context pool,* INFuse *produces the same result by its check fusion semantics, as existing constraint checking techniques do.*

Proof. Since the semantic structures of true value evaluation and link generation are highly consistent, we only give our proof when it comes to the truth value semantics. We here prove INFuse's checking correctness of truth value evaluation semantics for all seven formulas in detail.

Universal formula. We would rely on the checking correctness of ECC, Con-C, and PCC, and thus, we explain their truth value evaluation semantics for universal formula briefly here.

Let the universal formula be $\forall v \in C(f)$ and C contains m elements $(e_1, \ldots, e_m)$ after applying a context change chg. The truth value τ of the universal formula is defined as the conjunction of truth values $(t_1, \ldots, t_m)$ of subformula f for all elements in C. ECC evaluates each t_i in a sequential manner while Con-C evaluates each t_i concurrently. PCC considers the effect of chg, which can be split into four cases: (a) if chg did not affect the formula at all, each t_i would remain unchanged, as well as τ. (b) if chg added the element e_m into C, $t_1, \ldots, t_{m-1}$ would remain unchanged, and thus, τ would be the conjunction of its last value and t_m associated with e_m. (c) if chg deleted the element e_{m+1} from C, $t_1, \ldots, t_m$ would remain unchanged, and thus, τ would be the conjunction of them. (d) if chg affected another context related to f, then all t_i would need to be reevaluated partially in a similar manner.

We now analyze INFuse's truth value evaluation semantics for universal formula to prove its correctness. Firstly, the correctness of the entire semantics as shown in Fig. 7 is similarly guaranteed by the correctness of ECC's semantics due to their similarity. Secondly, Con-C's correctness confirms that evaluating truth values concurrently for independent elements can get the same results as evaluating serially, which guarantees the correctness of $\mathsf{eval}_{\mathsf{entire}}$ and $\mathsf{eval}_{\mathsf{partial}}$. Therefore, we only specifically analyze the correctness concerning cases of the partial semantics in Fig. 6:

- Case (1) is exactly the same as case (a) in PCC since it only focuses on whether the whole formula is affected.
- Case (2) extends the idea of case (b) in PCC to multiple context changes. These context changes only added elements $(y_1, \ldots, y_a)$ in C. Therefore, the last truth value (τ_0) is reusable according to case (b) in PCC. The correctness of new truth values $(t_1, \ldots, t_a)$ associated with new elements are guaranteed by $\mathsf{eval}_{\mathsf{entire}}$.
- Case (3) fuses the idea of case (b) and case (c) in PCC and extends to multiple context changes. Truth values associated with elements that were not deleted or updated by forthcoming context changes are reusable according to case (c) in PCC. The correctness of new truth values $(t_1, \ldots, t_{a+u})$ associated with new or updated elements are also guaranteed by $\mathsf{eval}_{\mathsf{entire}}$.
- Case (4) is exactly the same as case (d) in PCC, since it only focuses on whether subformula f is affected when C is not affected.
- Case (5) fuses the idea of case (b), case (c), and case (d) in PCC and extends to multiple context changes. The correctness of truth values $(t_1, \ldots, t_{a+u})$ associated with new elements or updated elements are guaranteed by $\mathsf{eval}_{\mathsf{entire}}$. Truth values $(t_{a+u+1}, \ldots, t_n)$ associated with elements that were not deleted or updated should be reevaluated partially since subformula f is affected according to case (d) in PCC. their correctness are guaranteed by $\mathsf{eval}_{\mathsf{partial}}$.

Existential formula. Since INFuse's truth value evaluation semantics for the existential formula is quite similar to that for the universal formula, the correctness of INFuse's truth value evaluation semantics for the existential formula can be proved follow the same procedure, i.e., the correctness of the entire semantics can be guaranteed by ECC's correctness, Con-C's correctness supports the $\mathsf{eval}_{\mathsf{entire}}$ and $\mathsf{eval}_{\mathsf{partial}}$ functions, and the partial semantics can be analyzed similarly.

`and`, `or`, and `implies` formulas. Fig. 12 shows the truth value evaluation semantics for `and` formula. The correctness of the entire semantics for `and` formula is trivial since it evaluates the truth value based on the logic of the formula. As for the partial semantics, every

$$\tau_{\text{entire}}[(f_1)\ \text{and}\ (f_2)]_\alpha = \tau_{\text{entire}}[f_1]_\alpha \wedge \tau_{\text{entire}}[f_2]_\alpha$$

(a) Entire semantics

$$\begin{aligned}
&\tau_{\text{partial}}[(f_1)\ \text{and}\ (f_2)]_\alpha = \\
&(1)\ \tau_0[(f_1)\ \text{and}\ (f_2)]_\alpha, \text{if } \mathsf{Affected}(f_1) = \mathsf{Affected}(f_2) = \mathsf{F}.\\
&(2)\ \tau_0[f_1]_\alpha \wedge \tau_{\text{partial}}[f_2]_\alpha, \text{if } \mathsf{Affected}(f_1) = \mathsf{F}, \mathsf{Affected}(f_2) = \mathsf{T}.\\
&(3)\ \tau_{\text{partial}}[f_1]_\alpha \wedge \tau_0[f_2]_\alpha, \text{if } \mathsf{Affected}(f_1) = \mathsf{T}, \mathsf{Affected}(f_2) = \mathsf{F}.\\
&(4)\ \tau_{\text{partial}}[f_1]_\alpha \wedge \tau_{\text{partial}}[f_2]_\alpha, \text{if } \mathsf{Affected}(f_1) = \mathsf{Affected}(f_2) = \mathsf{T}.
\end{aligned}$$

(b) Partial semantics

Fig. 12. INFuse's truth value evaluation semantics for `and` formula.

$$\tau_{\text{entire}}[(\text{not}\ (f)]_\alpha = \neg\tau_{\text{entire}}[f]_\alpha$$

(a) Entire semantics

$$\begin{aligned}
&\tau_{\text{partial}}[(\text{not}\ (f)]_\alpha = \\
&(1)\ \tau_0[\text{not}\ (f)]_\alpha,\ \text{if } \mathsf{Affected}(f) = \mathsf{F}.\\
&(2)\ \neg\tau_{\text{partial}}[f]_\alpha,\ \text{if } \mathsf{Affected}(f) = \mathsf{T}.
\end{aligned}$$

(b) Partial semantics

Fig. 13. INFuse's truth value evaluation semantics for `not` formula.

$$\tau_{\text{entire}}[bfunc(v_1, \cdots, v_n)]_\alpha = bfunc(v_1, \cdots, v_n)$$

(a) Entire semantics

$$\tau_{\text{partial}}[bfunc(v_1, \cdots, v_n)]_\alpha = \tau_0[bfunc(v_1, \cdots, v_n)]_\alpha$$

(b) Partial semantics

Fig. 14. INFuse's truth value evaluation semantics for $bfunc$ formula.

`and` formula has two subformulas, each of which could be affected by INFuse's arranged valid context changes. Therefore, INFuse partitions all situation into four cases. Besides, `or` and `implies` formulas can be proved in the same way.

`not` formula. Fig. 13 shows the truth value evaluation semantics for `not` formula. The entire semantics for `not` formula is straightforward and the partial semantics contain two cases since the subformula of `not` formula is either affected or not affected.

***bfunc* formula.** Fig. 14 shows INFuse's truth value evaluation semantics for $bfunc$ formula. $bfunc$ formula returns its result as we expect in the entire semantics and its last truth value is always reusable since it neither owns any subformula nor references any context.

Therefore, the correctness of truth value evaluation semantics for all seven formulas are proved, i.e., INFuse can achieve the same truth values as existing checking techniques. Moreover, the correctness of link generation semantics can be proved similarly, incurring that INFuse can achieve the same links as existing checking techniques. As a summary, INFuse can achieve the same inconsistency checking results as existing checking techniques. This completes our proof. □

In the following, we explain more realization details on both the WHAT-TO-CHECK part and the HOW-TO-CHECK part, and analyze its algorithmic complexity with comparisons to existing techniques.

3.4. INFUSE *realization details*

The preceding WHAT-TO-CHECK part decides a group of context changes that are valid to check together, and the HOW-TO-CHECK part guides how to complete the truth value evaluation and link generation for this group of context changes by fusing incremental and concurrent checking together. In the following, we explain more realization details in the two parts.

For the WHAT-TO-CHECK part, we explain how to enhance our task arrangement in practice. Recalling the preceding validity criterion in Definition 4, the basic idea is that any H-change should not follow an E-change within the same checking task, so as to avoid any missing inconsistency. In fact, as long as the inconsistency that may be hidden by the H-change has been reported in the last checking, this change can still be followed by any E-change without sacrificing the quality of checking results. Therefore, by temporarily buffering the context inconsistencies reported in the last checking, we can ignore such associated H-changes. Therefore, "chg_{new}" should be reexamined to be an H-change and at the same time do not relate to any element in the buffered inconsistencies (Line 9 in Algorithm 1), thus enhancing INFUSE's task arrangement by potentially enlarging more changes in a task in practice.

For the HOW-TO-CHECK part, we explain how to realize the check fusion concretely. Consider a constraint checking task whose included context changes have been decomposed into three sets (namely, $ASet$, $DSet$, and $USet$) for the task's associated consistency constraint. Recalling the preceding INFUSE's semantics in Figs. 8 and 11, the key point to start the check fusion is to first decide concurrent points in the constraint. Our intuition is three-folded: (1) the sub-tasks split at concurrent points should be balanced, and this requirement selects universal (∀) or existential (∃) formulas to be concurrent point candidates, since their subformulas correspond to identical formulas but with different variable-value bindings by definition, suggesting similar checking workloads (e.g., the example in Fig. 4); (2) each sub-task should contain sufficient checking workload, in order to avoid unnecessarily large concurrency management cost, and this requirement selects those higher-layer universal or existential formulas; (3) the finally decided concurrent points should be those affected by context changes (otherwise, their associated results can be reused according to the preceding partial checking semantics). We combine these three requirements into Algorithm 2, which eventually decides concurrent points to be those top-layer universal or existential formulas that are affected by context changes (i.e., involving at least one $ASet$, $DSet$, and $USet$, directly or indirectly, according to the preceding Affected function).

The algorithm analyzes a given consistency constraint s in a top-down manner, until it finds all necessary concurrent points that can cover all affected formulas inside this constraint. It starts from the root of the constraint, i.e., its top formula ($s.root$), and explores its subformula(s) to find those first encountered universal or existential formulas (Line 7) that are affected by context changes (Line 6). The exploration process must terminate since each terminal $bfunc$ is enclosed by at least one universal or existential formula. For example, we consider the two preceding constraint examples, whose tree-alike structures are illustrated in Fig. 2. For a constraint like in Fig. 2(a), if the and formula's both subformulas $\forall v_x \in C_x(f)$ and $\forall v_y \in C_y(f)$ are affected by context changes, then both of them are considered as concurrent points; otherwise, if only one subformula is affected, then it is the only concurrent point. For a constraint like in Fig. 2(b), if both C_x and C_y are affected by context changes, only the root formula $\forall v_x \in C_x(f)$ is considered as the concurrent point. Then we further consider our preceding constraint S_{loc} and its checking task $T = (chg_1, chg_2, chg_3, chg_4)$. Although both the universal formula (i.e., $\forall v_x \in C_x$) and the existential formula (i.e., $\exists v_y \in C_y$) are affected by these changes, INFUSE would select only the universal formula as the concurrent point.

Algorithm 2: Concurrent points selection

```
Input  : consistency constraint $s$
Output: set of $s$'s concurrent points $cpSet$
1  $cpSet = \emptyset$;
2  $stack$ = emptyStack();
3  $stack$.push($s.root$);
4  while $stack$ is not empty do
5      $f = stack$.pop();
6      if Affected($f$) == True then
7          if $f.type$ == ∀ or $f.type$ == ∃ then
8              $cpSet$.add($f$);
9          else if $f.type$ == and or $f.type$ == or or $f.type$ ==
           implies then
10             $stack$.push($f.left\_subformula$);
11             $stack$.push($f.right\_subformula$);
12         else if $f.type$ == not then
13             $stack$.push($f.subformula$);
14 return $cpSet$;
```

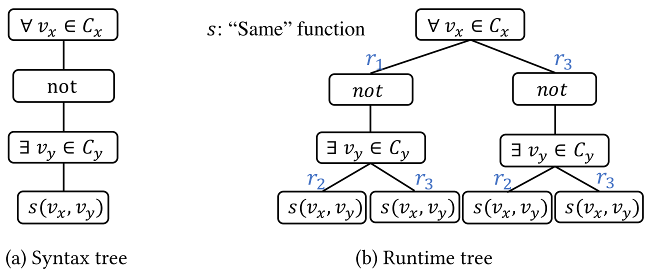


Fig. 15. Syntax and runtime tree examples for the S_{loc} constraint.

With concurrent points decided, INFUSE proceeds with its fusion checking, following the semantics in Figs. 6 and 9. INFUSE conducts the truth value evaluation and link generation according to encountered formula types and conditions (affected function and set value conditions) starting from the constraint's root formula in a top-down manner, to either invoke new calculations or reuse existing results (i.e., entire or partial checking). During this process, when invoking the eval or gen function in Figs. 8 and 11, INFUSE would decide whether to start concurrent checking according to whether the current formula is a previously decided concurrent point. If yes, INFUSE exploits the multi-threading support to assign each thread with a sub-task (i.e., checking the concerned subformula with a certain variable assignment). This naturally fuses concurrent checking into entire or partial checking. If no, INFUSE simply completes sub-tasks sequentially. When all sub-tasks are completed, their results are merged and propagated to the root formula, following the semantics in Figs. 8 and 11. We note that since upon a sub-task is assigned with a dedicated thread, no further splitting would be considered for this sub-task, this treatment makes INFUSE's fusion or incremental and concurrent checking simple and efficient.

3.5. INFUSE *complexity analyses*

In the following, we analyze how complex such a fusion checking behaves and how it is compared to existing incremental and concurrent checking algorithmically.

To facilitate our complexity analysis, we rely on two notions from the literature (Wang et al., 2021; Xu et al., 2010, 2013) for representing consistency constraints, namely, *syntax tree* and *runtime tree*. The former describes a constraint's structure in a hierarchical way, as illustrated in Fig. 15(a), representing our preceding constraint S_{loc} (other partial examples can be found in Fig. 2). The latter resembles the former except that it clones some sub-trees with different value assignments for variables introduced in universal or existential formulas, as illustrated in Fig. 15(b), where context C_x contains r_1 and r_3, and C_y contains r_2 and r_3.

We now analyze INFuse's HOW-TO-CHECK part, which dominates the whole computational complexity (the WHAT-TO-CHECK part consists of several simple runtime type checks only). Consider a given consistency constraint s, with its checking task consisting of some context changes. As aforementioned, INFuse decomposes the task into three sets (i.e., $ASet$, $DSet$, and $USet$) for each involved context in this constraint, and conducts the fusion checking with decided concurrent points. Let the number of context changes be m in this task and the height of constraint s be H. The height denotes the maximum hops from a constraint's syntax tree's root node to its leaf nodes, e.g., the height is three in Fig. 15(a). It is easy to observe that the task-to-set decomposition takes O(m) time, and that the concurrent-point decision takes O(H) time. In the following, we analyze in detail the complexity of the kernel fusion checking.

According to the preceding INFuse's checking realization, we analyze the complexity for completing the sub-task of each arranged thread starting at a concurrent point (named *concurrent cost*), and for merging and propagating intermediate results from concurrent points up to the root node (named *merge cost*). We have earlier noted that concurrent points are universal or existential formulas in a constraint, and thus they correspond to such nodes in the constraint's syntax tree. For example, considering constraint S_{loc} and its checking task $T = (chg_1, chg_2, chg_3, chg_4)$, the universal formula (i.e., $\forall v_x \in C_x$) is the only concurrent point as aforementioned, and thus the root node in the syntax tree (Fig. 15(a)) corresponds to this concurrent point. Besides, according to Algorithm 2, no other concurrent point would exist between a concurrent point and the root node. Therefore, for a syntax tree's corresponding runtime tree, its part from concurrent points to the root node would be exactly the same as that in the syntax tree. This brings two useful properties: (1) any concurrent point corresponds to a unique node in both the syntax tree and runtime tree, and (2) the hops from the root node to any concurrent point are no more than O(H), implying that the merge cost would be within O(H) time. Here we note that when analyzing the complexity of constraint checking upon a consistency constraint (fixed) given a sequence of context changes (not fixed), we are considering the impact of the number of these changes as well as their types. With this setting, the constraint itself never changes, and as such we can consider its height in the tree structure is a constant. Therefore, we here consider H as a constant, and reduce this cost to be O(1) time, while focusing on the main cost below.

This leaves us the main challenge of analyzing INFuse's complexity in completing the sub-task from a concurrent point. Let a considered concurrent point be c, and we analyze the averaged time complexity for completing its sub-task for one thread (all threads are concurrent).

We consider the sub-tree in constraint s's syntax tree with concurrent point c as the root node of this sub-tree. Let the height of this sub-tree be h, and it contains totally k universal or existential formula nodes, each associated with a specific context (named $ctx_0, \dots, ctx_{k-1}$). For ease of presentation, we let c's associated context be ctx_0, and the other $k-1$ contexts are ordered in a descending order on their heights (i.e., descending h_i for ctx_i, representing the hops from ctx_i to the lowest leaf node), as shown in Fig. 16.

To analyze the averaged complexity, we assume for all contexts in c's sub-tree, they: (1) are even distributed (i.e., with different locations and different heights), and (2) are even affected by context changes (i.e., with the same probability). For the former, the average height of all k contexts (except ctx_0) is half the sub-tree's height:

$$\frac{1}{k-1}\sum_{i=1}^{k-1} h_i = \frac{h}{2}. \tag{8}$$

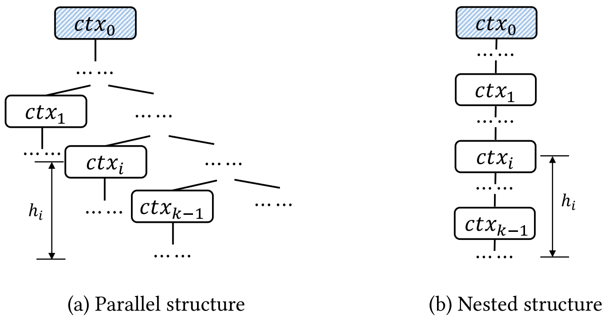


Fig. 16. Parallel and nested structures in a syntax tree.

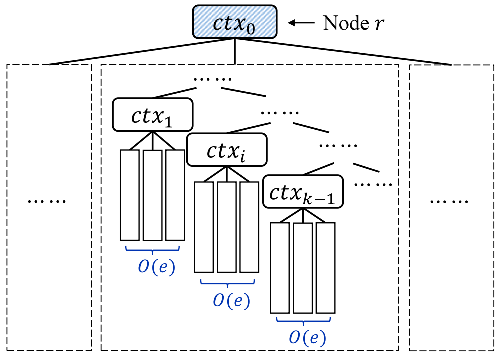


Fig. 17. Runtime tree of the parallel structure.

For the latter, since each context is affected by change (i.e., adding or deleting an element) always with the same probability, each context ($ctx_0, \dots, ctx_{k-1}$) contains the same number (say, e) of elements.

We note that different constraint structures have different impacts on the complexity analysis. Therefore, we consider two representative structures (i.e., *parallel* and *nested*), as illustrated in Fig. 16. For the former, a context is never within the scope of another context, while for the latter, a context is always within that of a previous context. We let node r be concurrent point c's corresponding node in constraint s's runtime tree, and the sub-tree with r as the root node contain n_0 nodes before applying context changes. Since c's associated context ctx_0 contains e elements, then each of r's sub-tree in the runtime tree contains $\frac{n_0}{e}$ nodes. In the following, we analyze for the two structures.

Parallel structure. Consider node r (concurrent point c's corresponding node in the runtime tree) with the parallel structure, as shown in Fig. 17.

In the parallel structure, one can approximate the total number of nodes in one of r's sub-trees also by accumulating each context ctx_i's sub-tree's node number (i.e., $O(e \cdot h_i)$). We thus have the following equation:

$$\sum_{i=1}^{k-1} O(e \cdot h_i) = O(\frac{n_0}{e}). \tag{9}$$

Then, to analyze the averaged time complexity of conducting INFuse's fusion checking for one of r's sub-trees (all sub-trees are checked concurrently by different threads), we consider three representative cases (i.e., only $ASet$, $DSet$, or $USet$ changes; other cases in between).

(1) *Only $ASet$ changes.* Recalling that each context is even affected by change, thus all contexts' corresponding $ASet$ should contain the elements for addition with a close magnitude. Let this number be $O(a)$. To conduct INFuse's fusion checking (i.e., truth value evaluation and link generation), INFuse needs to create $O(a)$ new sub-trees for node r that require entire checking ctx_0

affected, and adjust $O(e)$ sub-trees for node r that require partial checking (internal contexts also affected). To realize such creation and adjustment, INFuse assigns dedicated threads, one for each sub-tree of node r. Note that adding a new sub-tree is more time-consuming than adjusting an existing sub-tree since it requires three parts of jobs (i.e., node creation, truth value evaluation, and link generation), while adjusting an existing sub-tree requires only the latter two jobs. Therefore, we analyze the complexity of adding a new sub-tree to represent those for other sub-trees (since all are done concurrently, adding a new sub-tree represents the most complexity). For a newly created sub-tree (with $O(e+a)$ elements for each internal context, i.e., $\sum_{i=1}^{k-1}((e+a)\cdot h_i)$ nodes in total), each node would be visited three times for the node creation, truth value evaluation and link generation respectively. Therefore, the time cost is:

$$O(3\cdot\sum_{i=1}^{k-1}((e+a)\cdot h_i)). \quad (10)$$

Based on earlier derived Eq. (9), this can be reduced to:

$$O((3+\frac{3a}{e})\frac{n_0}{e}). \quad (11)$$

(2) *Only $USet$ changes.* Similarly, let the number of elements for update in $USet$ be $O(u)$. In this case, INFuse allocates $O(e)$ threads to update all $O(e)$ sub-trees of node r, in which $O(u)$ sub-trees require full updates (updating whole sub-trees), and the remaining $O(u)$ sub-trees require partial updates (updating parts affected by update changes to internal contexts). Similarly, as handling a full update is most time-consuming, we analyze its complexity to be representative. Note that all nodes ($O(\frac{n_0}{e})$ for each sub-tree) should be updated and visited twice, i.e., reevaluating truth values and regenerating links (no node creation required). Therefore, the time cost is:

$$O(2\cdot\frac{n_0}{e}). \quad (12)$$

(3) *Only $DSet$ changes.* Let the number of elements for deletion in $DSet$ be $O(d)$. In this case, INFuse allocates $O(e)$ threads for all $O(e)$ sub-trees of node r, where $O(d)$ sub-trees are whole deleted, and the remaining $O(e-d)$ sub-trees are adjusted (some internal parts are deleted). Similarly, handling an adjustment is most time-consuming, we analyze its complexity to be representative. For a sub-tree that requires an adjustment, INFuse needs to: (1) remove $O(d)$ branches for each context node, and (2) then reevaluate the truth value and regenerate links for each node on paths from a context node to node r. The former takes $O((k-1)\cdot d)$ time, and the latter takes $O(2\cdot\frac{1}{2}\cdot\sum_{i=1}^{k-1}(h-h_i))$ time, considering that all paths eventually merge into one in a random, steady way. Therefore, the combined time cost is:

$$O((k-1)\cdot d+2\cdot\frac{1}{2}\cdot\sum_{i=1}^{k-1}(h-h_i)). \quad (13)$$

Based on earlier derived Eq. (8) and Eq. (9), this can be reduced to:

$$O((\frac{2d}{eh}+\frac{1}{e})\frac{n_0}{e}). \quad (14)$$

Comparing the time costs for the three cases, we observe that cases (1) and (2) share a comparable complexity (coefficient is a small constant over one), while case (3) tends to be less complex (coefficient is smaller than one). Considering that in constraint checking, elements to be added, deleted, or updated for a given task typically occupy only a small proportion of all existing elements, we then have: O($a/u/d$)≪O(e). Therefore, we can conclude for the parallel structure that the $ASet$ case has the most time complexity, and $USet$ case has the slightly less time complexity, and the $DSet$ case has the least time complexity.

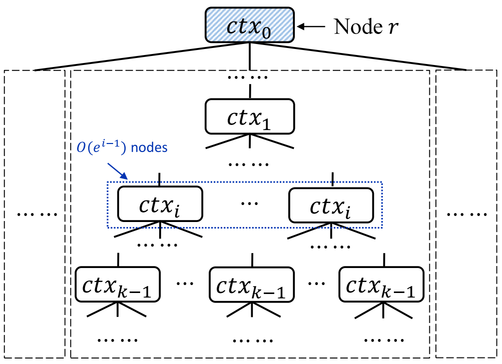


Fig. 18. Runtime tree for the nested structure.

Nested structure. We next consider node r (concurrent point c's corresponding node in the runtime tree) with the nested tree structure, as shown in Fig. 18. Similarly, we also assume $O(e)$ elements in each context, and this makes that each sub-tree from node r is continuously split into $O(e)$ branches upon each context node. Therefore, for context ctx_i, there would be $O(e^{i-1})$ corresponding context nodes in one of node r's sub-tree. To calculate the total number of nodes in one of node r's sub-trees, we accumulate to obtain this number by approximating a triangle-alike tree structure:

$$O(\frac{1}{2}\cdot h\cdot e^{k-1})=O(\frac{n_0}{e}). \quad (15)$$

Then, we similarly consider three cases:

(1) *Only $ASet$ changes.* In this case, similarly adding a whole sub-tree to node r would dominate the cost, and thus we analyze this to be representative. All nodes in such a sub-tree (with $a+e$ elements for each context) should be visited three times (for node creation, truth value evaluation, and link generation). Therefore, the time cost is:

$$O(3\cdot\frac{1}{2}\cdot h\cdot(e+a)^{k-1}). \quad (16)$$

Based on Eq. (15), it can be reduced to

$$O(3\cdot(1+\frac{a}{e})^{k-1}\frac{n_0}{e}). \quad (17)$$

(2) *Only $USet$ changes.* In this case, similarly fully updating a whole sub-tree to node r's would dominate the cost, and we analyze this. All nodes in such a sub-tree would be visited twice (node creation not required). Therefore, the time cost is:

$$O(2\cdot\frac{n_0}{e}). \quad (18)$$

(3) *Only $DSet$ changes.* In this case, similarly adjusting a whole sub-tree to node r would dominate the cost, and we analyze this. The time cost consists of two parts: (1) removing O(d) branches for each context node in this sub-tree, and (2) reevaluating truth values and regenerating link for nodes on paths from each context node to root node r. For the former, context ctx_i initially (before applying changes) corresponds to e^{i-1} context nodes in the sub-tree, and later (after applying changes) corresponds to $(e-d)^{i-1}$ context nodes. Then, with a typical top-down adjustment process, the time cost for this part is:

$$O(\sum_{i=1}^{k-1}d\cdot(e-d)^{i-1})=O(d\cdot\frac{(e-d)^{k-1}-1}{e-d-1}). \quad (19)$$

For the latter, all remaining nodes in the sub-tree ($e-d$ elements remaining now for each context now) except lowest-layer leaf node in the sub-tree should be visited twice for reevaluating truth values and regenerating links. Then, the time cost for this part is:

Table 2
Time complexity comparison.

Constraint structure	Checking technique	Set state		
		Only $ASet$ changes	Only $USet$ changes	Only $DSet$ changes
Parallel	ECC	$O(\frac{3ka(2e+a)^2}{4e} \cdot \frac{n_0}{e})$	$O(\frac{3ku(2e-u)^2}{2e} \cdot \frac{n_0}{e})$	$O(\frac{3kd(2e-d)^2}{4e} \cdot \frac{n_0}{e})$
	Con-C	$O(\frac{3ka(2e+a)}{2e} \cdot \frac{n_0}{e})$	$O(\frac{3ku(2e-u)}{e} \cdot \frac{n_0}{e})$	$O(\frac{3kd(2e-d)}{2e} \cdot \frac{n_0}{e})$
	PCC	$O(\frac{3a(2e+a)}{2e} \cdot \frac{n_0}{e})$	$O(\frac{3u(2e-u)}{e} \cdot \frac{n_0}{e})$	$O(\frac{d(2e-d)}{e} \cdot \frac{n_0}{e})$
	INFuse	$O(\frac{3(e+a)}{e} \cdot \frac{n_0}{e})$	$O(2 \cdot \frac{n_0}{e})$	$O(\frac{2d+h}{eh} \cdot \frac{n_0}{e})$
Nested	ECC	$O(\frac{3ka(2e+a)}{4}(\frac{e+a}{e})^{k-1} \cdot \frac{n_0}{e})$	$O(\frac{3ku(2e-u)}{2} \cdot \frac{n_0}{e})$	$O(\frac{3kd(2e-d)}{4} \cdot \frac{n_0}{e})$
	Con-C	$O(\frac{3ka}{2}(\frac{e+a}{e})^{k-1} \cdot \frac{n_0}{e})$	$O(3ku \cdot \frac{n_0}{e})$	$O(\frac{3kd}{2} \cdot \frac{n_0}{e})$
	PCC	$O(\frac{3(2e+a)}{2}(\frac{e+a}{e})^{k-1} \cdot \frac{n_0}{e})$	$O(\frac{4e^2-u^2}{e} \cdot \frac{n_0}{e})$	$O(\frac{(e+d)(2e-d)}{2e} \cdot \frac{n_0}{e})$
	INFuse	$O(3(\frac{e+a}{e})^{k-1} \cdot \frac{n_0}{e})$	$O(2 \cdot \frac{n_0}{e})$	$O(\frac{2d+2h}{(e-d-1)h}(1-\frac{d}{e})^{k-1}\frac{n_0}{e})$

$$O(2 \cdot \frac{1}{2} \cdot h \cdot (e-d)^{k-2}). \tag{20}$$

Combining the two parts, the total time cost is:

$$O(d \cdot \frac{(e-d)^{k-1}-1}{e-d-1} + h \cdot (e-d)^{k-2}). \tag{21}$$

Based on Eq. (15), this can be reduced to

$$O(\frac{2d+2h}{(e-d-1)h}(1-\frac{d}{e})^{k-1}\frac{n_0}{e}) \tag{22}$$

Considering O($a/u/d$)≪O(e), we can conclude for the nested structure that the $ASet$ case has the most time complexity (containing exponential calculation with a base over one) for INFuse, and the $USet$ and $DSet$ cases have a similar time complexity.

Finally, we similarly analyze the time complexities of existing constraint checking techniques (i.e., ECC (Nentwich et al., 2002), Con-C (Xu et al., 2013), and PCC (Xu et al., 2010)) for the comparison. Since these techniques check upon every single context change, we regard the three sets as three lists of context changes, i.e, $ASet$ responding to $O(k \cdot a)$ addition changes, $DSet$ responding to $O(k \cdot d)$ deletion changes, and $USet$ responding to $O(k \cdot u)$ deletion changes and following $O(k \cdot u)$ addition changes. To facilitate our analysis, we assume that the number of nodes in a sub-tree of node r and the number of sub-trees of node r evenly increase or decrease. Therefore, we measure their averages for estimating the average complexity of checking one single context change, and then multiply it with the number of context changes to estimate the overall time complexity of checking the three sets respectively. Following this idea, we adapt the time complexity analysis of existing checking techniques from their work (Xu et al., 2010, 2013), and give our analyzing results in Table 2 (we leave the full-length analyses to the Appendix for interested readers). As shown in Table 2, we can observe their relative differences in time complexity: generally, ECC is the most complex, Con-C and PCC are at the middle, and INFuse is the least complex.

Then combining all the analyses for the two extreme structures (parallel and nested), and the three set cases ($ASet$, $USet$, and $DSet$) for all checking techniques (ECC, PCC, Con-C, and our INFuse), we summarize our three main conclusions: (1) impact of the constraint structure: the parallel structure incurs the least complexity to constraint checking, and nested structure incurs the most complexity, and other mixed structures would behave in between; (2) impact of the set type: $ASet$ changes (context addition) incur the most complexity to constraint checking, $USet$ changes (context update) incur moderate complexity, and $DSet$ changes (context deletion) incur the least complexity; (3) the comparisons among all techniques: ECC has the weakest capability of handling complex constraint checking, Con-C/PCC has the moderate capability, and INFuse has the strongest capability. We would also validate these analyses in RQ4 in the evaluation.

4. Evaluation

In this section, we evaluate INFuse's performance and compare it with existing constraint checking techniques.

4.1. Research questions

We aim to answer the following three research questions:

- **RQ1 (Motivation)**: *How do existing constraint checking techniques behave when handling large-volume dynamic contexts?*
- **RQ2 (Effectiveness)**: *How effective is* INFuse *in constraint checking for detecting context inconsistencies, as compared with existing techniques?*
- **RQ3 (Fusion Effect)**: *How does* INFuse*'s fusion mechanism contribute to its efficiency improvement?*
- **RQ4 (Complexity Factor)**: *How is* INFuse*'s efficiency affected by different complexity factors?*
- **RQ5 (Practical Usage)**: *How effective is* INFuse *in constraint checking under real-life settings?*

4.2. Experimental design and setup

Application. For fair comparisons, we used the taxi application, SmartCity, as our experimental subject, following existing work (Xu et al., 2010, 2013, 2015; Wang et al., 2021). The application used massive taxi-driving data for smart route guidance.

Contexts. The application was accompanied with massive data concerning 2,716 vehicles monitored in a continuous period of 24 h, which include 4.3 million raw driving data lines (containing vehicle id, GPS coordinates, driving speed and orientation, and service status). These data correspond to 25.6 million *context changes* as modeled in the application. Fig. 19 illustrates the distribution of these context changes by 24 h-based groups (from 0 am–24 pm). We observed that these numbers can incur significantly varying workloads to constraint checking, since they range from 311,240 to 1,664,900 (up to a 435% difference). We believe that this characteristics can make our experimental data more representative for evaluating abilities of different constraint checking techniques against various workloads.

Constraints. We used all 48 consistency constraints associated with the application, also studied in existing work (Xu et al., 2010; Wang et al., 2021). They cover all formula types in the constraint language. They considered both spacial and temporal properties about vehicles' movements. These properties could be divided into four categories, namely, validating vehicles' geographical ranges, reasonable velocities, velocity-location relationships, and hot-area monitoring.

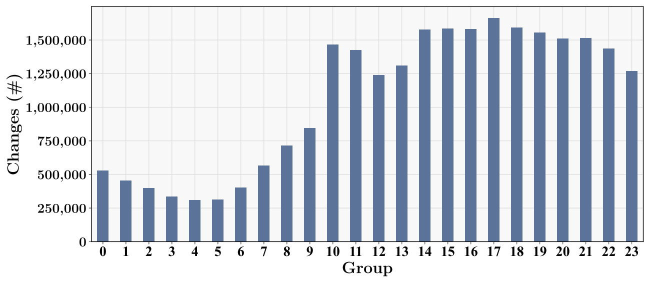


Fig. 19. Distribution of context changes for 24 h-based groups.

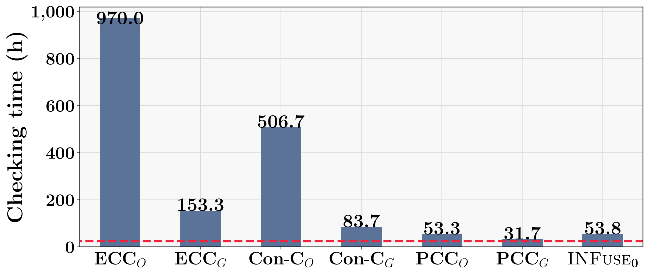


Fig. 20. Checking time comparison for the seven techniques with respect to all 24-h context changes (the red dashed line represents the 24-h time limit).

Process. In experiments, contexts are fed to the application with a middleware layer in between, which checks the contexts for consistency. We compared INFUSE with existing constraint checking techniques (ECC, PCC, and Con-C), using both their original versions (subscript "O") (Xu et al., 2010, 2013) and variants enhanced by GEAS (subscript "G") (Wang et al., 2021) for better scheduling for efficiency. We also compared INFUSE with a naïve implementation INFUSE$_0$ of the incremental-concurrent idea, which directly split incremental checking into parallel computing units (i.e., without INFUSE's concurrency maximization).

Setup. We design three *independent variables*:

- *Checking technique.* We compare eight techniques or variants, namely, ECC$_O$, ECC$_G$, Con-C$_O$, Con-C$_G$, PCC$_O$, PCC$_G$, INFUSE$_0$, and INFUSE.
- *Checking workload.* As aforementioned, different groups of context changes incur significantly varying workloads. Therefore, we use all 24 groups of context changes to evaluate and compare the performance of different constraint checking techniques (for fairness).
- *Running mode.* We study two running modes, namely, *offline* and *online*. With the former, next context changes are fed to the application only when previous changes have been handled (for comparing true efficiency differences). With the latter, context changes are fed to the application strictly according to their original timestamps and intervals in between, no matter whether previous changes have been handled or not (for testing in a real-life setting, possibly causing false negatives or positives).

We design three dependent variables:

- *Checking time.* It measures the total time spent on constraint checking.
- *Precision.* It measures the proportion of context inconsistencies that are correctly reported against all reported inconsistencies.
- *Recall.* It measures the proportion of context inconsistencies that are correctly reported against all inconsistencies that should be reported.

All experiments were conducted on a commodity PC with an AMD Ryzen 5600X 6-Core Processor with 32 GB RAM, installed with MS windows 10 Professional and Oracle Java 8.

To answer research question RQ1, we compare six existing constraint checking techniques and INFUSE$_0$ on all 24-h context changes under the offline mode to evaluate and compare their performance. To answer research question RQ2, we compare all eight constraint checking techniques (including INFUSE$_0$ and INFUSE) on all 24-h context changes, as well as 24 h-based groups separately under the offline mode, for evaluating and comparing their checking qualities (by reported inconsistencies) and efficiencies (by checking time). To answer research question RQ3, we study how INFUSE's fusion mechanism enhances the checking efficiency of incremental and concurrent techniques individually by selective enabling/disabling treatments in INFUSE in checking selected groups of context changes, and study how they are fused together to achieve INFUSE's overall efficiency improvement. To answer research question RQ4, we study INFUSE's checking efficiency by controlling different complexity factors, e.g., with different structures of consistency constraints (parallel or nested) and different set types in checking tasks ($ASet$, $USet$, or $DSet$). To answer research question RQ5, we compare all eight constraint checking techniques on 24 h-based groups under the online mode (i.e., with real-life timestamps and intervals), for evaluating and comparing their checking qualities (by precision and recall) and efficiencies (by checking time).

4.3. Experimental results

We answer the five research questions in turn.

4.3.1. RQ1 (Motivation)

We compared the checking time of the seven constraint checking techniques on all 24-h context changes in Fig. 20.

We observe that the checking time varied significantly for different constraint checking techniques, e.g., ECC up to 153.3–970.0 h, Con-C for 83.7–506.7 h, and PCC for 31.7–53.3 h. We note that the time limit for handling all 24-h context changes is 24 h, as illustrated by the red dashed line, and thus none of these techniques fulfilled this requirement, e.g., the worst case of ECC$_O$ took more than 40 days! This strongly calls for more efficient constraint checking techniques. Besides, also as INFUSE$_0$ shows, directly splitting incremental checking into parallel computing units did not bring significant improvement (53.8 h), behaving even worse than PCC (31.7–53.3 h).

Therefore, we answer RQ1 as follows: *All existing constraint checking techniques and naïve implementation of the incremental-concurrent idea failed to deliver required checking efficiency, calling for new research efforts.*

4.3.2. RQ2 (Effectiveness)

We then compared the checking time of INFUSE to the other seven techniques on all 24-h context changes in Fig. 21. As the comparison was under the offline mode, all context changes were fed and then checked in turn, and thus all checking techniques obtained correct inconsistency detection results (this may not be true for the online mode, as discussed later). Therefore, we focus on the checking time comparison here.

From Fig. 21, we observe that INFUSE took only eight hours to complete constraint checking for all 24-h context changes, which satisfied the aforementioned time limit requirement (note that none of the other seven techniques succeeded, as discussed in RQ1). Moreover, we also observe that INFUSE brought significant efficiency improvement, as compared with other constraint checking techniques, e.g., 18.2x–120.3x efficiency improvement against ECC (or 94.8%–99.2% checking time reduction), 9.5x–62.3x improvement against Con-C (or 90.4%–98.4% time reduction), 3.0x–5.7x improvement against PCC (or 74.8%–85.0% time reduction), and 5.7x improvement against

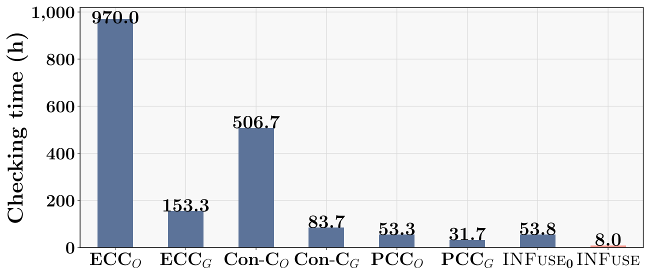


Fig. 21. Checking time comparison for all the eight checking techniques with respect to all 24-h context changes.

INFUSE_0 (or 85.1% time reduction). This shows INFUSE's general superiority and stable high-efficiency for large-volume constraint checking tasks. Note that INFUSE's clear efficiency improvement over existing checking techniques also echoes our earlier conclusion (3) in Section 3.5.

To further evaluate INFUSE's effectiveness across different workloads, we next compared the checking time of all the eight constraint checking techniques on 24 h-based groups in both Fig. 22 (in linear ordinate coordinates) and Fig. 23 (in logarithmic ordinate coordinates) for better illustration and comparisons. We observe that: (1) Although different workloads incurred greatly varying checking time (from seconds to hours, hundreds even thousands of times in the performance difference), INFUSE behaved consistently significant and stable efficiency improvement for checking all groups of context changes, against all other techniques. For example, INFUSE's efficiency improvement for the lightest workload (group 4 in the time slot of 4 am–5 am) is 0.0x–18.6x and that for the heaviest workload (group 17 in the time slot of 5 pm–6 pm) is 3.1x–166.2x. The average efficiency improvement for 24 h-based groups is 2.3x–98.1x, as compared with other techniques. (2) INFUSE's checking time (from 3.4 s to 0.8 h) satisfied all one-hour time limits for every group, consistently exhibiting INFUSE's high efficiency across different checking workloads. (3) When comparing INFUSE with the naïve implementation INFUSE_0, their difference was also large and impressive, e.g., the time reduction varying from 32.2% to 85.6%. We owe all these achievements to INFUSE's carefully designed concurrency maximization and fusion soundness as explained and analyzed earlier.

We also studied the trend of INFUSE's efficiency improvements for 24 h-based groups with the increasing workloads in Fig. 24. Note that the number of context changes to handle in each hour largely approximates the checking workload. In the figure, we observe that with the growth of the checking workload, INFUSE's efficiency improvement over the other existing checking techniques and INFUSE_0 generally hold a stably increasing trend. This strongly suggests INFUSE's potential in handling even higher checking workloads.

Therefore, we answer RQ2 as follows: INFUSE *worked significantly efficiently, achieving 3.0x–120.3x improvements, as compared with all other constraint checking techniques. Besides,* INFUSE *worked stably and were suitable for higher checking workloads.*

4.3.3. RQ3 (Fusion effect)

We then study how INFUSE's fusion mechanism contributes to its efficiency improvement. Generally, INFUSE infuses two typical constraint checking techniques, i.e., incremental and concurrent checking, together. However, as studied in RQ1, directly combining them can lead to efficiency sacrifice instead, i.e., INFUSE_0's efficiency is even worse than PCC_O. By proposing its task arrangement in WHAT-TO-CHECK and fusion treatment in HOW-TO-CHECK, INFUSE succeeds in soundly fusing incremental and concurrent checking together, with promising efficiency (3.0x–120.3x efficiency improvements) as studied in RQ2. To further study how INFUSE's fusion mechanism contributes to such efficiency improvement, we design two INFUSE's variants, $\text{INFUSE}_{\text{incre}}$ and $\text{INFUSE}_{\text{con}}$. $\text{INFUSE}_{\text{incre}}$ disables the concurrent treatment in INFUSE and retains incremental checking with INFUSE's fusion mechanism, while $\text{INFUSE}_{\text{con}}$ disables the incremental treatment in INFUSE and retains concurrent checking with INFUSE's fusion mechanism. We took ECC_O as the baseline (i.e., set as 1) and compared relative efficiency improvements for the other five techniques (i.e., PCC_O, Con-C_O, INFUSE, and its two variants) over ECC_O on group 9 of context changes (median workload). Results are shown in Fig. 25.

From the figure, we observe that the efficiency improvement of INFUSE_0 over ECC_O (16.5x) is even smaller than that of PCC_O (17.5x), echoing that combining incremental and concurrency checking directly actually compromises the checking efficiency, also earlier observed in RQ1. However, when such incremental and concurrent techniques are supported by INFUSE's fusion mechanism, i.e., $\text{INFUSE}_{\text{incre}}$ and $\text{INFUSE}_{\text{con}}$, their efficiency would be largely improved, i.e., from the original 17.5x to 39.1x for PCC, and from 1.1x to 12.5x for Con-C, suggesting great contributions of INFUSE's fusion mechanism to both further improving the original incremental and concurrent superiority. This is mainly because for PCC, INFUSE's fusion mechanism significantly amplifies its computational intensity (i.e., how many nodes in the tree structures are computed in each scheduled constraint checking), which is well above that of PCC, as illustrated by the intensity distribution in Fig. 26(a). This explains how $\text{INFUSE}_{\text{incre}}$ outperforms PCC. For Con-C, INFUSE's fusion mechanism brings more potentials for concurrent checking, as illustrated by the thread number distribution in Fig. 26(b). We observe that INFUSE's median thread number (101) is well above that (68) of Con-C. This explains how $\text{INFUSE}_{\text{con}}$ outperforms Con-C. Altogether, INFUSE's fusion mechanism can further improve both PCC's and Con-C's high efficiency. After combining them together (i.e., INFUSE), we can observe significantly more efficiency improvement, i.e., 105.2x as compared to ECC_O. Compared to directly combining incremental and concurrent checking by INFUSE_0, INFUSE's fusion mechanism can indeed make extra and dominant contributions.

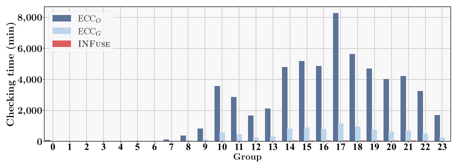
Therefore, we answer RQ3 as follows: INFUSE*'s fusion mechanism contributes greatly to its impressive efficiency improvement on constraint checking, by significantly enhancing the efficiency of its fused incremental and concurrent checking.*

4.3.4. RQ4 (Complexity factor)

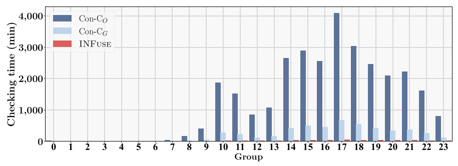
To investigate the impacts of different complexity factors, we study INFUSE's checking efficiency under different structures of consistency constraints (parallel or nested) and different set types in checking tasks ($ASet$, $USet$ or $DSet$).

First, we investigate how INFUSE's efficiency was affected by different constraint structures (parallel vs. nested). We measured INFUSE's average checking time when checking each of its arranged tasks against a parallel constraint (from category "geographical ranges") and nested consistency constraint (from category "reasonable velocities", "velocity-location relationships", or "hot-area monitoring"). Results are shown in Fig. 27. We observe that INFUSE spent significantly more time on checking consistency constraints with nested structures (5.02 ms on average) than those with parallel structures (0.08 ms on average), with a difference around 62.8x. This suggests that nested structures can incur obviously heavier checking workloads, echoing our conclusion (1) in Section 3.5.

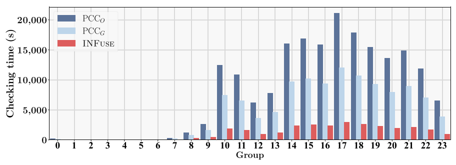
Then, to investigate how INFUSE's efficiency was affected by different set types of INFUSE's arranged checking tasks, we control to check $ASet$, $DSet$, and $USet$ tasks individually with an increasing set size. We simulated elements in each set with randomly synthesized values. To be more realistic, we randomly selected ten snapshots referring to ten different checking timepoints to apply INFUSE from the checking process conducted in RQ2. We used them as the initial statuses before checking and applying INFUSE to check the controlled tasks including a non-empty $ASet$, $DSet$, or $USet$ with a increasing set sizes (from 1 to 16, following the average set size during the whole checking



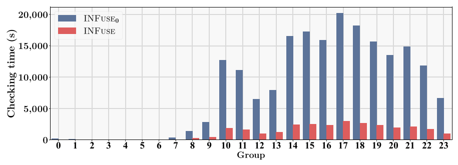
(a) Checking time comparison for ECC and INFuse (unit: minute)



(b) Checking time comparison for Con-C and INFuse (unit: minute)

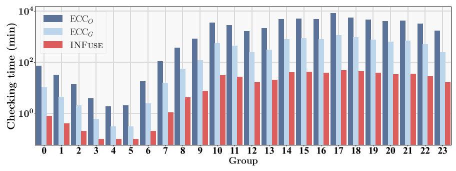


(c) Checking time comparison for PCC and INFuse (unit: second)

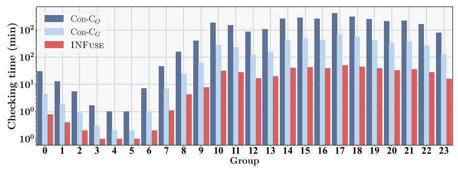


(d) Checking time comparison for INFUSE$_0$ and INFuse (unit: second)

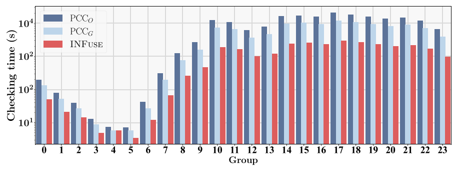
Fig. 22. Checking time comparison for all checking techniques on 24 h-based groups (linear scale).



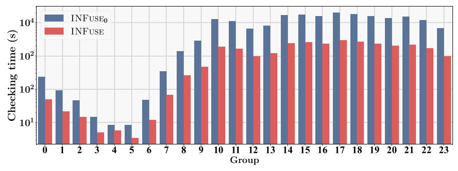
(a) Checking time comparison for ECC and INFuse (unit: minute)



(b) Checking time comparison for Con-C and INFuse (unit: minute)



(c) Checking time comparison for PCC and INFuse (unit: second)



(d) Checking time comparison for INFUSE$_0$ and INFuse (unit: second)

Fig. 23. Checking time comparison for all checking techniques on 24 h-based groups (logarithmic scale).

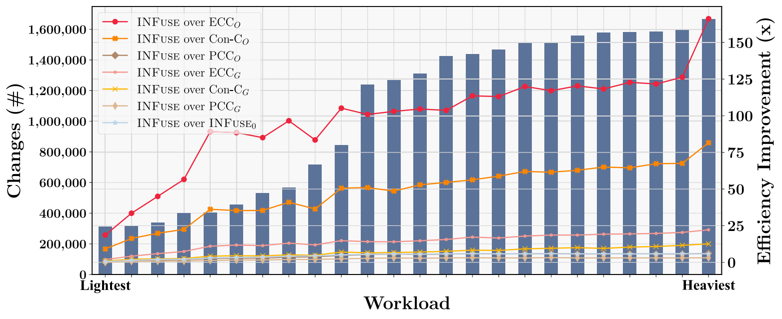


Fig. 24. INFuse's efficiency improvement over existing checking techniques on 24 h-based groups (sorted by increasing workloads).

process in RQ2). As shown in Fig. 28, we can observe that INFuse spent significantly more checking time on checking $ASet$ tasks than $USet$ and $DSet$ tasks (ratio is about 100:50:1), suggesting that checking $ASet$ indeed induces the most checking workloads for INFuse, while $USet$ incurs the median workloads and $DSet$ incurs the least. With the increasing set size, INFuse followed an almost linear growing trend in the checking time. This also echoes our conclusion (2) in Section 3.5.

Therefore, we can answer RQ4 as follows: *Both complex constraint structures (e.g., nested) and checking sets ($ASet$) can incur the most checking workloads for* INFuse*, confirming our complexity analyses in Section* 3.5.

Table 3
Comparisons among all techniques under the online mode.

Checking techniques	Metrics	Groups																							
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
ECC$_O$	T_{cost}(min)	57.5	33.9	14.8	4.4	2.1	2.3	18.4	59.1	61.7	64.2	66.8	62.3	60.6	64.1	61.0	64.8	64.2	67.2	65.5	65.1	65.9	60.7	60.9	62.6
	Precision(%)	4.4%	100.0%	100.0%	100.0%	100.0%	100.0%	15.3%	26.6%	7.0%	8.0%	8.4%	7.3%	7.0%	6.4%	7.4%	7.7%	9.0%	7.8%	8.3%	8.0%	8.1%	8.1%	8.4%	6.1%
	Recall(%)	4.1%	100.0%	100.0%	100.0%	100.0%	100.0%	15.2%	23.1%	3.2%	3.4%	1.5%	1.5%	1.9%	1.6%	1.4%	1.4%	1.6%	1.3%	1.4%	1.4%	1.6%	1.4%	1.5%	1.7%
Con-C$_O$	T_{cost}(min)	29.9	13.0	5.8	1.8	0.9	1.0	7.1	42.7	60.2	59.9	61.0	63.1	60.7	59.8	62.5	60.7	64.3	62.5	63.5	63.3	61.5	63.3	60.8	60.3
	Precision(%)	4.8%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	75.9%	7.0%	6.3%	7.0%	6.4%	5.7%	5.6%	6.3%	7.1%	7.3%	7.0%	7.2%	7.2%	6.6%	7.0%	6.8%	5.0%
	Recall(%)	4.8%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	74.7%	5.4%	4.2%	2.1%	2.0%	2.1%	2.2%	1.7%	1.7%	2.0%	1.6%	1.8%	1.9%	1.9%	1.9%	1.9%	2.2%
PCC$_O$	T_{cost}(min)	3.4	1.9	0.8	0.4	0.2	0.2	0.8	5.8	19.4	25.2	56.6	57.0	58.5	58.7	56.9	56.7	57.0	56.8	56.9	57.2	56.4	56.4	56.6	58.4
	Precision(%)	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	92.5%	4.2%	3.8%	3.9%	5.8%	4.3%	4.4%	4.2%	4.6%	4.6%	4.5%	4.4%	4.7%	4.4%	3.8%
	Recall(%)	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	90.3%	3.6%	3.5%	3.8%	5.6%	3.6%	3.4%	3.5%	3.5%	3.7%	3.6%	3.7%	3.8%	3.9%	3.8%
ECC$_G$	T_{cost}(min)	11.0	5.3	2.2	0.9	0.4	0.5	2.7	16.4	55.1	59.8	57.7	57.5	58.3	58.6	57.2	58.4	58.2	59.6	58.8	57.8	57.2	56.9	57.6	58.0
	Precision(%)	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	35.5%	7.0%	4.7%	4.1%	4.1%	3.6%	4.7%	5.1%	5.4%	5.5%	5.4%	5.3%	4.9%	4.8%	4.7%	3.8%
	Recall(%)	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	35.4%	6.6%	3.2%	3.0%	3.6%	3.0%	2.8%	2.7%	3.2%	2.7%	3.0%	3.0%	3.1%	2.8%	3.1%	3.3%
Con-C$_G$	T_{cost}(min)	4.6	2.2	0.9	0.5	0.3	0.3	1.1	6.8	24.3	32.5	57.3	57.2	59.3	58.6	56.3	55.7	56	56.3	56.6	56.1	56.4	56.4	57.2	59.7
	Precision(%)	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	92.9%	3.8%	3.6%	3.8%	3.8%	3.8%	4.4%	4.2%	4.6%	4.8%	4.5%	4.0%	4.4%	4.3%	3.3%
	Recall(%)	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	90.0%	3.5%	3.4%	3.5%	3.7%	3.2%	3.3%	3.5%	3.2%	3.8%	3.7%	3.4%	3.6%	3.8%	3.3%
PCC$_G$	T_{cost}(min)	2.6	1.5	0.7	0.3	0.2	0.2	0.6	4.1	13.0	18.3	59.9	60.0	52.2	54.7	59.9	59.9	60.2	59.9	60.0	59.9	59.7	60.1	59.9	54.3
	Precision(%)	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	91.0%	3.9%	3.8%	4.6%	33.0%	4.4%	4.0%	4.2%	4.1%	4.2%	4.4%	4.0%	4.6%	4.5%	4.9%
	Recall(%)	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	91.0%	3.9%	3.8%	4.6%	33.0%	4.4%	3.8%	4.1%	4.0%	4.3%	4.2%	4.1%	4.4%	4.3%	4.8%
INFUSE$_O$	T_{cost}(min)	3.5	2.0	1.0	0.4	0.2	0.2	0.9	6.2	21.0	26.5	57.0	57.2	58.7	58.0	57.2	57.0	57.5	57.7	56.9	56.8	57.0	57.2	57.2	58.6
	Precision(%)	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	92.6%	4.1%	3.9%	3.9%	6.2%	4.3%	4.4%	4.3%	4.5%	4.6%	4.4%	4.3%	4.5%	4.4%	3.7%
	Recall(%)	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	90.4%	3.5%	3.6%	3.9%	3.8%	5.6%	3.3%	3.6%	3.4%	3.7%	3.6%	3.5%	3.7%	3.8%	3.6%
INFUSE	T_{cost}(min)	0.9	0.8	0.4	0.2	0.2	0.2	0.3	1.6	4.0	7.6	28.7	27.5	16.2	19.7	39.7	42.0	38.5	49.2	43.6	38.8	32.7	34.8	27.8	16.1
	Precision(%)	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	98.2%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%
	Recall(%)	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	98.2%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	10.0%	100.0%	100.0%	100.0%	100.0%

🟥 🟧 ⬜ and 🟦 represent the precision or the recall is [0.0%, 10.0%), [10.0%, 90.0%), [90.0%, 100.0%), and 100.0% respectively.

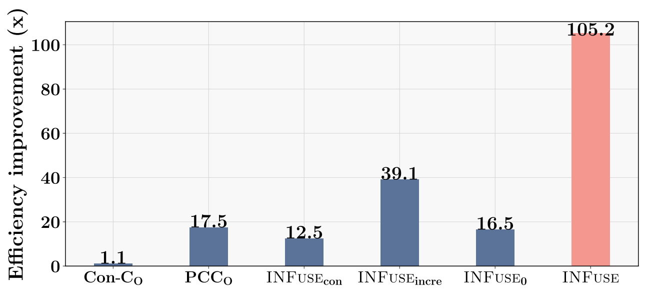


Fig. 25. Efficiency improvement comparison of six checking techniques over ECC$_O$.

4.3.5. RQ5 (Practical usage)

We also compared INFuse with the other seven techniques under an online mode, which simulated real-life context change scenarios with actual timestamps and intervals. We focus on the checking quality (by precision and recall) and efficiency (by checking time). Table 3 lists the comparison results.

From the table, we observe that: (1) All six existing checking techniques (ECC, Con-C, PCC, and their variants) are undesirable because they were all subject to quality problems. Consider the most efficient existing checking technique PCC$_G$. It produced satisfactory checking results (precision = 100% and recall = 100%) only for the first 9 groups (i.e., group 0 to group 8) among all 24 groups (these 9 groups represents the least workloads). Then, its quality declined rapidly for other higher-workload groups, i.e., it suffered from extremely severe quality problems (below 10% precision and recall) for 13 groups (i.e., from group 10 to group 23 except group 13). Regarding other existing techniques, since they were even less efficient than PCC$_G$, they produced much worse results, e.g., PCC$_O$ and Con-C$_G$ suffered from such extremely severe quality problems for 14 groups (58% of all 24 groups), ECC$_G$ for 15 groups (63%), and ECC$_O$ and Con-C$_O$ even for 17 groups (71%). This exactly motivates us for a desirable constraint checking technique like INFuse, as we studied in this work. (2) The naïve implementation INFuse$_0$ also could not alleviate the quality problems. On one hand, it still suffered from such quality problems for 14 groups. On the other hand, as compared to PCC$_G$, INFuse$_0$ exhibited even less efficiently by taking more checking time for groups in which they both reported the same correct inconsistency results, thus reflecting their true efficiency difference since all context changes are fairly checked in this case. This again echoes our claim that directly splitting incremental checking into parallel computing units would easily compromise the efficiency instead. (3) INFuse both obtained proper constraint checking results and achieved high checking efficiency. For all 24 h-based groups, INFuse achieved a 100% precision and recall for 23 groups except group 10, for which INFuse achieved a 98.2% precision and recall, significantly higher than those of other techniques (precision down to 3.3% and recall down to 1.3%). We note that a 100% precision and recall may not always be possible since network connection and object serialization costs were inevitable under real-life settings, which could affect other key computations unexpectedly. Regarding the checking efficiency, INFuse always took the least time for all 24 groups, 12.5%–98.4% less than other techniques across different groups.

Therefore, we answer RQ5 as follows: INFuse *worked significantly efficiently under real-life dynamic scenarios with a 100% precision and recall for almost all groups, while other techniques could suffer down to a 3.3% precision and 1.3% recall, exhibiting* INFuse*'s clear technical superiority and applicability.*

4.4. Threats analysis and discussion

First, in our experiments, we selected only one application, and this could cause possible threats to experimental conclusions. Regarding this, we have tried to alleviate such threats by carefully considering relevant factors: (1) The application was also used in existing work (Wang et al., 2021; Xu et al., 2010, 2013, 2015), with the same set of consistency constraints and contexts, so as to facilitate across-technique comparisons (to be fair); (2) We used all 48 consistency constraints associated with the application, which cover all constraints used in existing work's experiments (to be comprehensive), and these constraints also cover all formula types supported in the constraint language (to be complete); (3) All 24 groups of context changes (collected in a continuous period of 24 h) were selected from a whole day to represent varying and realistic workloads, for examining the effectiveness of different constraint checking techniques (to be representative); (4) All the constraint checking techniques were repeated for every context change group around five times (to be reliable), except for ECC$_O$ and Con-C$_O$, which ran extremely too costly (each run lasted over 40 and 20 days, respectively).

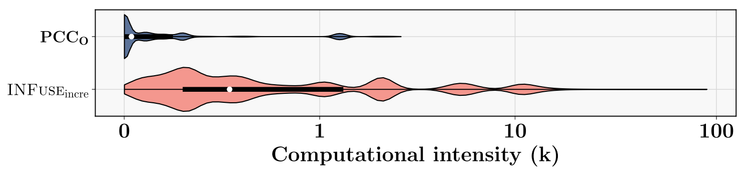
Second, to avoid possible platform and implementation bias, we (re)implemented all constraint checking techniques under the same I/O interfaces and data structures according to their respective publications, and compared with their released versions for ensuring the correctness of our implementation. We have also checked all the inconsistencies reported by every constraint checking technique. We have also released our implementation[1] to facilitate follow-up research.

5. Related work

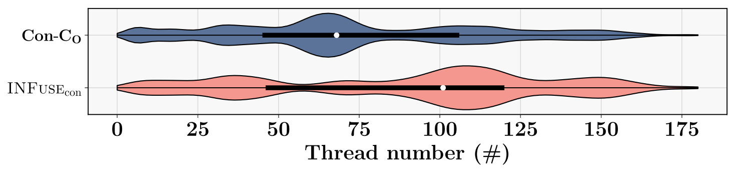
In this section, we discuss the related work in recent years, following four aspects, namely, managing consistency for software artifacts, reducing noises in raw low-quality data, detecting inconsistencies for application contexts, and resolving detected context inconsistencies. These four aspects closely relate to our studied context inconsistency problem in this work.

Managing consistency for software artifacts. Our software engineering community has extensively studied the problem of consistency management for various software artifacts, which concern different software development processes, e.g., software refactoring (Le et al., 2017), method name suggestion (Li et al., 2021), agile model-based development (Jongeling et al., 2019), and the whole software engineering process (Mayr-Dorn et al., 2021a). Some pieces of work focus on managing the consistency of traditional software artifacts, like edit scripts (Kehrer et al., 2013), UML models (Bashir et al., 2016; Messaoudi et al., 2017; Wei and Sun, 2021), XML documents (Nentwich et al., 2002; Reiss, 2006; Handley et al., 2021), and distributed source code (Demuth et al., 2016), which are featured as being typically static or evolving slowly. This line of work mainly pays attention to the effectiveness of consistency management rather than efficiency. Other pieces of work tackle more dynamic artifacts in context-aware systems (Limón et al., 2018; Chen et al., 2022a), attention-aware systems (Limón et al., 2019), and safety-critical systems (Mayr-Dorn et al., 2021b). These systems recently receive increasing attention for their functional qualities, and we are working along this line with extensive application scenarios, like Pollen Wise (pollen Sense, 2022), Humanoid Companion Robot (Kuo et al., 2021), self-driving vehicle systems (Waymo, 2022; Davies, 2017), and unmanned aerial vehicles (UAVs) (Yoon and Noh, 2022; Mazumdar et al., 2022; Lahmeri et al., 2022). Unlike traditional software artifacts, these artifacts are featured as changing rapidly, thus requiring more efficient consistency management. Our work in this article studies consistency of application contexts, which are modeled at the application layer based on its perceived environmental conditions with some derivation processes from raw data. For such applications, some frameworks or middleware infrastructures, like Cabot (Xu et al., 2004), Adam (Xu et al., 2012), Lime (Murphy et al., 2006), and CARISMA (Capra et al., 2003), have

[1] https://github.com/yuzi-zly/INFUSE



(a) Computational intensity distribution for PCC$_O$ and INFuse$_{incre}$ (logarithm axis)



(b) Thread number distribution for Con-C$_O$ and INFuse$_{con}$

Fig. 26. Distribution comparisons for studying INFuse's fusion mechanism.

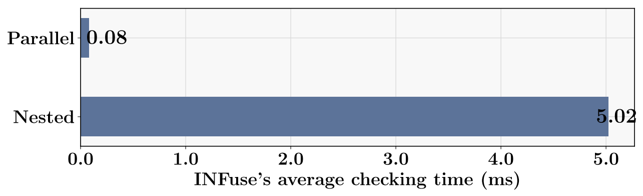


Fig. 27. INFuse's average checking time for parallel and nested structure.

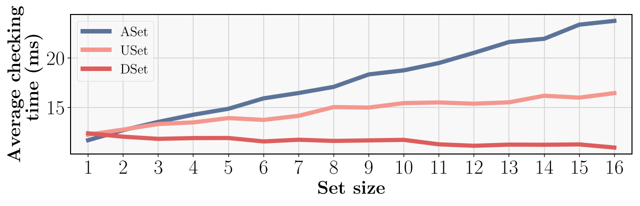


Fig. 28. INFuse's checking time comparison for $ASet$, $USet$, and $DSet$.

also been developed to specially support context-aware properties with quality guarantees (e.g., consistency or reliability).

Reducing noises in raw low-quality data. Raw environmental data for applications are mainly collected by various physical sensors (e.g., cameras and microphones). Typically, raw data contain natural noises due to the instability of sensor readings. For example, the widely used radio-based object identification and tracking RFID technology can often be subject to missing or cross reading problems (Jeffery et al., 2006; Rao et al., 2006; Patil et al., 2015; Fescioglu-Ünver et al., 2015). To reduce such noises, one line of work set pre-specific filtering thresholds (Rao et al., 2006) or matching patterns (Chaudhuri et al., 2003) to filter data to make them meet specific requirements. However, one major limitation of such techniques is that they focus mainly on proper threshold selection or pattern designing, while sometimes application developers or system administrators are not fully aware of application quality requirements. Recently, another line of work took advantage of artificial intelligence (AI) technologies to clean raw data for better quality. For example, Darcy et al. (2011) proposed a methodology to combine highly intelligent feature set definition and classifying techniques to handle false-positive data problems. Wei et al. (2022) proposed to select training data accurately facing the larger number of the noisy labels in the datasets. Chun et al. (2019) and Tan et al. (2019) proposed to reduce data noises for UAVs via convolutional neural networks. However, such data cleaning techniques would require substantial training data and may not be easy to adapt to other dynamic application scenarios.

Detecting inconsistencies for application contexts. This aspect focuses on how to efficiently and effectively detect inconsistencies in dynamic application contexts. On this particular aspect, various techniques work with varying efficiency gains and costs. For example, xlinkit (Nentwich et al., 2002) worked in a full checking way, as the correctness baseline, to detect all possible inconsistencies in artifacts under checking; PCC (Xu et al., 2010) checked incrementally by reusing previous results for higher efficiency; Con-C (Xu et al., 2013) checked concurrently on parallel computational units with similar workloads. All these techniques are useful for different application requirements, but are gradually becoming less effective, with the rapid growth of environmental dynamics and context volume. Regarding this, GEAS (Wang et al., 2021) was proposed to adaptively schedule the checking of multiple context changes together to help accelerate a spectrum of existing techniques. Our work resembles this line of efforts, but builds on dynamic validity criteria derived from incremental and concurrent checking, different from GEAS, which builds only on static constraint information. As a result, INFuse works even more efficiently than any existing constraint checking technique, either originally or combined with GEAS, as our experimental results reported. Besides, Xu et al. (2007a) theoretically analyzed possible link generation wastes in constraint checking, which opened a new direction to further improve the checking efficiency (i.e., reducing the link generation part rather than making the detection itself faster). Chen et al. (2022b) worked along this line, and recently went further by proposing to analyze and generate exactly necessary-only links (i.e., eliminating all redundant link generation), and this effort can additionally help improve the efficiency for context inconsistency detection.

Resolving detected context inconsistencies. Besides detecting inconsistencies for application contexts, one relevant and important aspect of research efforts is around resolving detected context inconsistencies. Existing inconsistency resolution work can be roughly classified into two categories. One category of work proposed various resolution techniques based on heuristics. For example, Chomicki et al. (2003) selected a random context for removal to solve the inconsistency among multiple contexts to minimize the cost. Bu et al. (2006) removed all contexts related with the same inconsistency to play safety. Xu et al. (2008) proposed another heuristic technique, which removed contexts participated in the detected inconsistencies more frequently to balance the cost and safety. However, these techniques could unexpectedly cause applications to behave abnormally, since they may accidentally remove important contexts applications are relying on. The other category of work took application logics into consideration during to the fixing process for the detected inconsistencies. For example, Chen et al. (2011) proposed to resolve inconsistencies with the help of application semantics to maximize possible application workflows. Xu et al. (2007b, 2011) and Khelladi et al. (2019) proposed to analyze and minimize side effects of such fixing or resolution actions unexpectedly on applications. These pieces of research efforts are consequent actions after high-efficient context inconsistency detection, as we studied in this work, for a large-spectrum of adaptive modern applications.

6. Conclusion

In this work, we studied the context inconsistency detection problem, and analyzed how to substantially boost its efficiency over state-of-the-art techniques. We proposed a novel INFuse approach, which on one hand automatically identifies valid and maximized context change groups for concurrency maximization, and on the other hand soundly fuses incremental and concurrent checking together for reuse and efficiency maximization. These efforts work on both the constraint checking aspect and checking scheduling aspect, thus outperforming any existing constraint checking technique and checking scheduling strategy, as well as their combinations, realizing a 3.0x–120.3x efficiency improvement with desirable quality guarantees. In future, we plan to more extensively validate INFuse on other application scenarios with massive context data, and explore further finer-granularity tuning strategies inside the fusion checking for unexpectedly dynamic checking workloads, making it more general and applicable.

CRediT authorship contribution statement

Lingyu Zhang: Conceptualization, Software, Validation, Writing – original draft. **Huiyan Wang:** Methodology, Writing – review & editing, Formal analysis, Funding acquisition. **Chuyang Chen:** Investigation, Writing – review & editing. **Chang Xu:** Resources, Writing, Supervision, Funding acquisition. **Ping Yu:** Investigation, Funding acquisition.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Chang Xu, Ping Yu report financial support was provided by Natural Science Foundation of China. Huiyan Wang reports financial support was provided by Leading-edge Technology Program of Jiangsu Natural Science Foundation.

Data availability

Data will be made available on request.

Acknowledgments

This work was supported by the Natural Science Foundation of China under Grant Nos. 61932021 and 62072225, and the Leading-edge Technology Program of Jiangsu Natural Science Foundation under Grant Nos. BK20202001 and BK20220771. The authors would also like to thank the support from the Fundamental Research Funds for the Central Universities of China (020214380102 and 020214912220), and Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

Appendix

This appendix is to complement our main article with more details on INFuse's fusion checking and time complexity analyses of different checking techniques. In the following, we first give definitions for necessary functions and operators, and then elaborate on the checking semantics for INFuse's truth value evaluation and link generation (for other formula types, not discussed in the main article). In the end, we give time complexity analyses for existing checking techniques.

A.1. Functions and operators

We define necessary functions and operators below.

A.1.1. Affected function

As aforementioned, we define the Affected function to indicate whether a formula itself or its nested subformula is affected by the context changes given in a constraint checking task. Consider a specific formula inside a consistency constraint. The Affected function returns T (meaning True), if and only if the formula itself or any of its contained subformula(s) references a context involved in any $ASet$, $DSet$, or $USet$ associated with this constraint; otherwise, it returns F (meaning False). Formally,

- Affected($\forall/\exists v \in C(f)$) = T, if $ASet \neq \emptyset$ or $DSet \neq \emptyset$ or $USet \neq \emptyset$ or Affected(f) = T; otherwise, F.
- Affected((f_1) `and/or/implies` (f_2)) = T, if Affected(f_1) = T or Affected(f_2) = T; otherwise, F.
- Affected(`not` (f)) = T, if Affected(f) = T; otherwise, F.
- Affected($bfunc(v_1, v_2, \ldots, v_n)$) = F.

A.1.2. Flip and FlipSet functions

We define the Flip function to reverse a link's linkType without changing the link's variable assignments, and the FlipSet function is used to apply the Flip function to each link in a link set. Formally,

- Flip(violated, variable assignments) = (satisfied, variable assignments).
- Flip(satisfied, variable assignments) = (violated, variable assignments).
- FlipSet(S) = {Flip(l) | $l \in S$}.

A.1.3. Type and assignments functions

We define the Type and Assignments functions to retrieve a link's specific linkType and variable assignments information from a given link, respectively, i.e.,

- Type(l) = l.linkType.
- Assignments(l) = l.variable assignments.

A.1.4. Concatenate function and ⊗ operator

We define the Concatenate function to combine two links with the same linkType into a new link, consisting of this linkType and the union of all concerned variable assignments from the two links. Further, the ⊗ operator concatenates two link sets by applying the Concatenate function to the link pairs formed by every link from set S_1 and every link from set S_2, i.e.,

- Concatenate(l_1, l_2) = (Type(l_1), Assignments(l_1) ∪ Assignments(l_2)).
- $S_1 \otimes S_2$ = {Concatenate(l_1, l_2) | $l_1 \in S_1 \wedge l_2 \in S_2$}, if $S_1 \neq \emptyset$ and $S_2 \neq \emptyset$; otherwise, $S_1 \cup S_2$.

A.2. Truth value evaluation

In the following, we give INFuse's truth value evaluation semantics for the ∃, `or`, and `implies` formulas (we have earlier introduced the semantics for other formula types, i.e., ∀, `and`, `not`, and $bfunc$, in Section 3.3).

A.2.1. Existential formula, i.e., $\exists v \in C(f)$

Figs. 29 and 30 give INFuse's entire and partial truth value evaluation semantics for the existential formula. Similar to that for the universal formula we discussed earlier, this semantics also invokes $\text{eval}_{\text{entire}}$ or $\text{eval}_{\text{partial}}$ functions (shown in Fig. 31) to calculate truth values for subformula f concerning different elements.

$$\tau_{\text{entire}}[\exists v \in C(f)]_\alpha = \mathsf{F} \vee \tau_{\text{entire}}[f]_{\text{bind}((v,x_1),\alpha)} \vee \cdots \vee \tau_{\text{entire}}[f]_{\text{bind}((v,x_n),\alpha)} | x_i \in C$$

Fig. 29. INFuse's entire truth value evaluation semantics for the existential formula.

$$\begin{aligned}
&\tau_{\text{partial}}[\exists v \in C(f)]_\alpha = \\
&(1)\ \tau_0[\exists v \in C(f)]_\alpha,\ \text{if Affected}(f) = \mathsf{F} \text{ and } (ASet = \emptyset \text{ and } DSet = \emptyset \text{ and } USet = \emptyset). \\
&(2)\ \tau_0[\exists v \in C(f)]_\alpha \vee t_1 \vee \cdots \vee t_a, \text{where } (t_1, \cdots, t_a) = \text{eval}_{\text{entire}}(\tau[f]_{\text{bind}((v,y_j),\alpha)} \mid y_j \in ASet), \\
&\ \text{if Affected}(f) = \mathsf{F} \text{ and } (ASet \neq \emptyset \text{ and } DSet = \emptyset \text{ and } USet = \emptyset). \\
&(3)\ \mathsf{F} \vee \tau_0[f]_{\text{bind}((v,x_1),\alpha)} \vee \cdots \vee \tau_0[f]_{\text{bind}((v,x_{n-a-u}),\alpha)} \vee t_1 \vee \cdots \vee t_{a+u} \mid x_i \in C \setminus (ASet \cup USet)), \\
&\quad \text{where } (t_1, \cdots, t_{a+u}) = \text{eval}_{\text{entire}}(\tau[f]_{\text{bind}((v,y_j),\alpha)} \mid y_j \in ASet \cup USet), \\
&\ \text{if Affected}(f) = \mathsf{F} \text{ and } (DSet \neq \emptyset \text{ or } USet \neq \emptyset). \\
&(4)\ \mathsf{F} \vee t_1 \vee \cdots \vee t_n, \text{where } (t_1, \cdots, t_n) = \text{eval}_{\text{partial}}(\tau[f]_{\text{bind}((v,x_i),\alpha)} \mid x_i \in C), \\
&\ \text{if Affected}(f) = \mathsf{T} \text{ and } (ASet = \emptyset \text{ and } DSet = \emptyset \text{ and } USet = \emptyset). \\
&(5)\ \mathsf{F} \vee t_1 \vee \cdots \vee t_n, \text{where } (t_1, \cdots, t_{a+u}) = \text{eval}_{\text{entire}}(\tau[f]_{\text{bind}((v,y_j),\alpha)} \mid y_j \in ASet \cup USet) \\
&\quad \text{and } (t_{a+u+1}, \cdots, t_n) = \text{eval}_{\text{partial}}(\tau[f]_{\text{bind}((v,x_i),\alpha)} \mid x_i \in C \setminus (ASet \cup USet)), \\
&\ \text{if Affected}(f) = \mathsf{T} \text{ and } (ASet \neq \emptyset \text{ or } DSet \neq \emptyset \text{ or } USet \neq \emptyset).
\end{aligned}$$

Fig. 30. INFuse's partial truth value evaluation semantics for the existential formula.

$$\begin{aligned}
&\text{eval}_{\text{entire}}(\tau[f]_{\text{bind}((v,x_i),\alpha)} \mid x_i \in Set) = \\
&\quad (1)\ \tau_{\text{entire}}[f]_{\text{bind}((v,x_1),\alpha)} \parallel \cdots \parallel \tau_{\text{entire}}[f]_{\text{bind}((v,x_s),\alpha)}, \\
&\qquad \text{if } \exists v \in C(f) \text{ is a concurrent point;} \\
&\quad (2)\ \tau_{\text{entire}}[f]_{\text{bind}((v,x_1),\alpha)}\ ;\ \cdots\ ;\ \tau_{\text{entire}}[f]_{\text{bind}((v,x_s),\alpha)}, \\
&\qquad \text{otherwise.}
\end{aligned}$$

$$\begin{aligned}
&\text{eval}_{\text{partial}}(\tau[f]_{\text{bind}((v,x_i),\alpha)} \mid x_i \in Set) = \\
&\quad (1)\ \tau_{\text{partial}}[f]_{\text{bind}((v,x_1),\alpha)} \parallel \cdots \parallel \tau_{\text{partial}}[f]_{\text{bind}((v,x_s),\alpha)}, \\
&\qquad \text{if } \exists v \in C(f) \text{ is a concurrent point;} \\
&\quad (2)\ \tau_{\text{partial}}[f]_{\text{bind}((v,x_1),\alpha)}\ ;\ \cdots\ ;\ \tau_{\text{partial}}[f]_{\text{bind}((v,x_s),\alpha)}, \\
&\qquad \text{otherwise.}
\end{aligned}$$

Fig. 31. Semantics of the eval functions (entire and partial checking).

$$\tau_{\text{entire}}[(f_1) \text{ or } (f_2)]_\alpha = \tau_{\text{entire}}[f_1]_\alpha \vee \tau_{\text{entire}}[f_2]_\alpha$$

$$\tau_{\text{entire}}[(f_1) \text{ implies } (f_2)]_\alpha = \neg\tau_{\text{entire}}[f_1]_\alpha \vee \tau_{\text{entire}}[f_2]_\alpha$$

Fig. 32. INFuse's entire truth value evaluation semantics for `or` and `implies` formulas.

A.2.2. `or` and `implies` formulas , i.e., (f_1) `or/implies` (f_2)

Fig. 32 gives INFuse's entire truth value evaluation semantics for the two formulas. Similar to the `and` formula, `or` and `implies` formulas reference no direct context, and we only need to consider the Affected function on their subformulas f_1 and f_2. Incremental evaluation would be applied to the affected subformulas, as shown in Fig. 33.

A.3. Link generation

In the following, we give INFuse's link generation semantics for other formulas not discussed earlier (i.e., ∃, `and`, `or`, `implies`, `not`, *bfunc*), while the ∀ formula has been introduced Section 3.3.

A.3.1. Existential formula , i.e., $\exists v \in C(f)$

$$\begin{aligned}
&\tau_{\text{partial}}[(f_1) \text{ or } (f_2)]_\alpha = \\
&(1)\ \tau_0[(f_1) \text{ or } (f_2)]_\alpha, \text{if Affected}(f_1) = \text{Affected}(f_2) = \mathsf{F}. \\
&(2)\ \tau_0[f_1]_\alpha \vee \tau_{\text{partial}}[f_2]_\alpha, \text{if Affected}(f_1) = \mathsf{F}, \text{Affected}(f_2) = \mathsf{T}. \\
&(3)\ \tau_{\text{partial}}[f_1]_\alpha \vee \tau_0[f_2]_\alpha, \text{if Affected}(f_1) = \mathsf{T}, \text{Affected}(f_2) = \mathsf{F}. \\
&(4)\ \tau_{\text{partial}}[f_1]_\alpha \vee \tau_{\text{partial}}[f_2]_\alpha, \text{if Affected}(f_1) = \text{Affected}(f_2) = \mathsf{T}.
\end{aligned}$$

$$\begin{aligned}
&\tau_{\text{partial}}[(f_1) \text{ implies } (f_2)]_\alpha = \\
&(1)\ \tau_0[(f_1) \text{ implies } (f_2)]_\alpha, \text{if Affected}(f_1) = \text{Affected}(f_2) = \mathsf{F}. \\
&(2)\ \neg\tau_0[f_1]_\alpha \vee \tau_{\text{partial}}[f_2]_\alpha, \text{if Affected}(f_1) = \mathsf{F}, \text{Affected}(f_2) = \mathsf{T}. \\
&(3)\ \neg\tau_{\text{partial}}[f_1]_\alpha \vee \tau_0[f_2]_\alpha, \text{if Affected}(f_1) = \mathsf{T}, \text{Affected}(f_2) = \mathsf{F}. \\
&(4)\ \neg\tau_{\text{partial}}[f_1]_\alpha \vee \tau_{\text{partial}}[f_2]_\alpha, \text{if Affected}(f_1) = \text{Affected}(f_2) = \mathsf{T}.
\end{aligned}$$

Fig. 33. INFuse's partial truth value evaluation semantics for `or` and `implies` formulas.

Figs. 34 and 35 give INFuse's entire and partial link generation semantics for the existential formula. Similar to that for the universal formula, it also invokes the $\text{gen}_{\text{entire}}$ and $\text{gen}_{\text{partial}}$ functions (shown in Fig. 36) to generate links for subformula f concerning different elements.

A.3.2. `and` , `or`, and `implies` formulas , i.e., (f_1) `and/or/implies` (f_2)

For ease of understanding, we take the `and` formula as an example to explain the principles in its link generation:

- If both f_1 and f_2 are evaluated to true, they together decide the satisfaction of this `and` formula. Then, the ⊗ operator is used to generate links that explain the formula's satisfaction.
- If both f_1 and f_2 are evaluated to false, either of them can decide the violation of this `and` formula. Then, the union of links from f_1 and f_2 explains the formula's violation.
- If one subformula is evaluated to true and the other is evaluated to false, then the latter can decide the violation of this `and` formula. Then, links coming from the latter explain the formula's violation.

$$\mathcal{L}_{\text{entire}}[\exists v \in C(f)]_{\alpha} = \{l \mid l \in \{(\text{satisfied}, \{(v, x_i)\})\} \otimes \mathcal{L}_{\text{entire}}[f]_{\text{bind}((v,x_i),\alpha)}\} \mid x_i \in C \wedge \tau[f]_{\text{bind}((v,x_i),\alpha)} = \text{T}).$$

Fig. 34. INFuse's entire link generation semantics for the existential formula.

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

(1) $\mathcal{L}_0[\exists v \in C(f)]_{\alpha}$, if $\text{Affected}(f) = \text{F}$ and ($ASet = \emptyset$ and $DSet = \emptyset$ and $USet = \emptyset$).

(2) $\mathcal{L}_0[\exists v \in C(f)]_{\alpha} \cup (\{(\text{satisfied}, \{v, y_1\})\} \otimes l_1) \cup \cdots \cup (\{(\text{satisfied}, \{v, y_{a'}\})\} \otimes l_{a'})$,
where $(l_1, \cdots, l_{a'}) = \text{gen}_{\text{entire}}(\mathcal{L}[f]_{\text{bind}((v,y_j),\alpha)} \mid y_j \in ASet \wedge \tau[f]_{\text{bind}((v,y_j),\alpha)} = \text{T})$,
if $\text{Affected}(f) = \text{F}$ and ($ASet \neq \emptyset$ and $DSet = \emptyset$ and $USet = \emptyset$).

(3) $(\{(\text{satisfied}, \{v, y_1\})\} \otimes l_1) \cup \cdots \cup (\{(\text{satisfied}, \{v, y_{a'+u'}\})\} \otimes l_{a'+u'}) \cup$
$\{l \mid l \in \{(\text{satisfied}, \{(v, x_i)\})\} \otimes \mathcal{L}_0[f]_{\text{bind}((v,x_i),\alpha)}\} \mid x_i \in C \setminus (ASet \cup USet) \wedge \tau[f]_{\text{bind}((v,x_i),\alpha)} = \text{T}$,
where $(l_1, \cdots, l_{a'+u'}) = \text{gen}_{\text{entire}}(\mathcal{L}[f]_{\text{bind}((v,y_j),\alpha)} \mid y_j \in ASet \cup USet \wedge \tau[f]_{\text{bind}((v,y_j),\alpha)} = \text{T})$,
if $\text{Affected}(f) = \text{F}$ and ($DSet \neq \emptyset$ or $USet \neq \emptyset$).

(4) $\emptyset \cup (\{(\text{satisfied}, \{v, x_1\})\} \otimes l_1) \cup \cdots \cup (\{(\text{satisfied}, \{v, x_{n'}\})\} \otimes l_{n'})$,
where $(l_1, \cdots, l_{n'}) = \text{gen}_{\text{partial}}(\mathcal{L}[f]_{\text{bind}((v,x_i),\alpha)} \mid x_i \in C \wedge \tau[f]_{\text{bind}((v,x_i),\alpha)} = \text{T})$,
if $\text{Affected}(f) = \text{T}$ and ($ASet = \emptyset$ and $DSet = \emptyset$ and $USet = \emptyset$).

(5) $\emptyset \cup (\{(\text{satisfied}, \{v, y_1\})\} \otimes l_1) \cup \cdots \cup (\{(\text{satisfied}, \{v, y_{n'}\})\} \otimes l_{n'})$,
where $(l_1, \cdots, l_{a'+u'}) = \text{gen}_{\text{entire}}(\mathcal{L}[f]_{\text{bind}((v,y_j),\alpha)} \mid y_j \in ASet \cup USet \wedge \tau[f]_{\text{bind}((v,y_j),\alpha)} = \text{T})$
and $(l_{a'+u'+1}, \cdots l_{n'}) = \text{gen}_{\text{partial}}(\mathcal{L}[f]_{\text{bind}((v,x_i),\alpha)} \mid x_i \in C \setminus (ASet \cup USet) \wedge \tau[f]_{\text{bind}((v,x_i),\alpha)} = \text{T})$,
if $\text{Affected}(f) = \text{T}$ and ($ASet \neq \emptyset$ or $DSet \neq \emptyset$ or $USet \neq \emptyset$).

Fig. 35. INFuse's partial link generation semantics for the existential formula.

$\text{gen}_{\text{entire}}(\mathcal{L}[f]_{\text{bind}((v,x_i),\alpha)} | x_i \in Set \wedge \tau[f]_{\text{bind}((v,x_i),\alpha)} = \text{T})$

(1) $\mathcal{L}_{\text{entire}}[f]_{\text{bind}((v,x_1),\alpha)} \parallel \cdots \parallel \mathcal{L}_{\text{entire}}[f]_{\text{entire}((v,x_s),\alpha)}$,
if $\exists v \in C(f)$ is a concurrent point.

(2) $\mathcal{L}_{\text{entire}}[f]_{\text{bind}((v,x_1),\alpha)}\ ;\ \cdots\ ;\ \mathcal{L}_{\text{entire}}[f]_{\text{bind}((v,x_s),\alpha)}$,
otherwise.

$\text{gen}_{\text{partial}}(\mathcal{L}[f]_{\text{bind}((v,x_i),\alpha)} \mid x_i \in Set \wedge \tau[f]_{\text{bind}((v,x_i),\alpha)} = \text{T})$

(1) $\mathcal{L}_{\text{partial}}[f]_{\text{bind}((v,x_1),\alpha)} \parallel \cdots \parallel \mathcal{L}_{\text{partial}}[f]_{\text{bind}((v,x_s),\alpha)}$,
if $\exists v \in C(f)$ is a concurrent point.

(2) $\mathcal{L}_{\text{partial}}[f]_{\text{bind}((v,x_1),\alpha)}\ ;\ \cdots\ ;\ \mathcal{L}_{\text{partial}}[f]_{\text{bind}((v,x_s),\alpha)}$,
otherwise.

Fig. 36. Semantics of the gen functions (entire and partial checking).

The principles for the `or` and `implies` formulas are similar. We thus give INFuse's entire link generation semantics for these three formulas in Fig. 37.

Similar to INFuse's truth value evaluation semantics for the three formulas, INFuse can also conduct incremental link generation according to the Affected function on subformulas f_1 and f_2. We similarly give INFuse's partial link generation semantics for the `and`, `or`, and `implies` formulas in Fig. 38, Fig. 39, and Fig. 40 respectively.

`not` and $bfunc$ formulas , i.e, `not` (f) *and* $bfunc(v_1, \ldots, v_n)$

Fig. 41 gives INFuse's entire link generation semantics for the `not` and $bfunc$ formulas. For the `not` formula, it inverts the linkType of links coming from its subformula f. For the $bfunc$ formula, it always generates an empty link set since the links that contain variables in the $bfunc$ formula are supposed to be generated where these variables are defined (i.e., at upper-layer universal and existential formulas). Fig. 42 gives INFuse's partial link generation semantics for the two formulas. For the `not` formula, the Affected function on its subformula f would internally decide the reusability of its previously generated links. The $bfunc$ formula would still generate an empty link set.

A.4. Time complexity analysis

In the following, we give the time complexity analysis of existing checking techniques. We use the same notations in Section 3.4 in our main article so that we can reuse some analysis results. As mentioned in our main article, our base idea is that we assume that the number of nodes in one sub-tree of node r and the number of sub-trees of node r both increase or decrease evenly so that we can use their averages to estimate the time complexity for one single context change, and then estimate the overall time complexity by multiplying the number of context changes.

Specifically, to estimate the averaged time complexity for one single context change, we need to know: (1) the average number of nodes in one sub-tree of node r per context change (let it be N), (2) the average number of added or removed nodes in one sub-tree of node r per context change (let it be Δ), (3) the average number of updated

$\mathcal{L}_{\text{entire}}[(f_1) \text{ and } (f_2)]_\alpha =$

(1) $\mathcal{L}_{\text{entire}}[f_1]_\alpha \otimes \mathcal{L}_{\text{entire}}[f_2]_\alpha$, if $\tau[f_1]_\alpha = \tau[f_2]_\alpha = \mathsf{T}$.

(2) $\mathcal{L}_{\text{entire}}[f_1]_\alpha \cup \mathcal{L}_{\text{entire}}[f_2]_\alpha$, if $\tau[f_1]_\alpha = \tau[f_2]_\alpha = \mathsf{F}$.

(3) $\mathcal{L}_{\text{entire}}[f_2]_\alpha$, if $\tau[f_1]_\alpha = \mathsf{T}, \tau[f_2]_\alpha = \mathsf{F}$.

(4) $\mathcal{L}_{\text{entire}}[f_1]_\alpha$, if $\tau[f_1]_\alpha = \mathsf{F}, \tau[f_2]_\alpha = \mathsf{T}$.

$\mathcal{L}_{\text{entire}}[(f_1) \text{ or } (f_2)]_\alpha =$

(1) $\mathcal{L}_{\text{entire}}[f_1]_\alpha \cup \mathcal{L}_{\text{entire}}[f_2]_\alpha$, if $\tau[f_1]_\alpha = \tau[f_2]_\alpha = \mathsf{T}$.

(2) $\mathcal{L}_{\text{entire}}[f_1]_\alpha \otimes \mathcal{L}_{\text{entire}}[f_2]_\alpha$, if $\tau[f_1]_\alpha = \tau[f_2]_\alpha = \mathsf{F}$.

(3) $\mathcal{L}_{\text{entire}}[f_1]_\alpha$, if $\tau[f_1]_\alpha = \mathsf{T}, \tau[f_2]_\alpha = \mathsf{F}$.

(4) $\mathcal{L}_{\text{entire}}[f_2]_\alpha$, if $\tau[f_1]_\alpha = \mathsf{F}, \tau[f_2]_\alpha = \mathsf{T}$.

$\mathcal{L}_{\text{entire}}[(f_1) \text{ implies } (f_2)]_\alpha =$

(1) $\mathsf{FlipSet}(\mathcal{L}_{\text{entire}}[f_1]_\alpha) \otimes \mathcal{L}_{\text{entire}}[f_2]_\alpha$, if $\tau[f_1]_\alpha = \mathsf{T}, \tau[f_2]_\alpha = \mathsf{F}$.

(2) $\mathsf{FlipSet}(\mathcal{L}_{\text{entire}}[f_1]_\alpha) \cup \mathcal{L}_{\text{entire}}[f_2]_\alpha$, if $\tau[f_1]_\alpha = \mathsf{F}, \tau[f_2]_\alpha = \mathsf{T}$.

(3) $\mathcal{L}_{\text{entire}}[f_2]_\alpha$, if $\tau[f_1]_\alpha = \tau[f_2]_\alpha = \mathsf{T}$.

(4) $\mathsf{FlipSet}(\mathcal{L}_{\text{entire}}[f_1]_\alpha)$, if $\tau[f_1]_\alpha = \tau[f_2]_\alpha = \mathsf{F}$.

Fig. 37. INFuse's entire link generation semantics for `and`, `or`, and `implies` formulas.

$\mathcal{L}_{\text{partial}}[(f_1) \text{ and } (f_2)]_\alpha =$

(1) $\mathcal{L}_0[(f_1) \text{ and } (f_2)]_\alpha$, if $\mathsf{Affected}(f_1) = \mathsf{Affected}(f_2) = \mathsf{F}$.

(2) a. $\mathcal{L}_{\text{partial}}[f_1]_\alpha \otimes \mathcal{L}_0[f_2]_\alpha$, if $\tau[f_1]_\alpha = \tau[f_2]_\alpha = \mathsf{T}$.

b. $\mathcal{L}_{\text{partial}}[f_1]_\alpha \cup \mathcal{L}_0[f_2]_\alpha$, if $\tau[f_1]_\alpha = \tau[f_2]_\alpha = \mathsf{F}$.

c. $\mathcal{L}_0[f_2]_\alpha$, if $\tau[f_1]_\alpha = \mathsf{T}, \tau[f_2]_\alpha = \mathsf{F}$.

d. $\mathcal{L}_{\text{partial}}[f_1]_\alpha$, if $\tau[f_1]_\alpha = \mathsf{F}, \tau[f_2]_\alpha = \mathsf{T}$.

if $\mathsf{Affected}(f_1) = \mathsf{T}, \mathsf{Affected}(f_2) = \mathsf{F}$.

(3) a. $\mathcal{L}_0[f_1]_\alpha \otimes \mathcal{L}_{\text{partial}}[f_2]_\alpha$, if $\tau[f_1]_\alpha = \tau[f_2]_\alpha = \mathsf{T}$.

b. $\mathcal{L}_0[f_1]_\alpha \cup \mathcal{L}_{\text{partial}}[f_2]_\alpha$, if $\tau[f_1]_\alpha = \tau[f_2]_\alpha = \mathsf{F}$.

c. $\mathcal{L}_{\text{partial}}[f_2]_\alpha$, if $\tau[f_1]_\alpha = \mathsf{T}, \tau[f_2]_\alpha = \mathsf{F}$.

d. $\mathcal{L}_0[f_1]_\alpha$, if $\tau[f_1]_\alpha = \mathsf{F}, \tau[f_2]_\alpha = \mathsf{T}$.

if $\mathsf{Affected}(f_1) = \mathsf{F}, \mathsf{Affected}(f_2) = \mathsf{T}$.

(4) a. $\mathcal{L}_{\text{partial}}[f_1]_\alpha \otimes \mathcal{L}_{\text{partial}}[f_2]_\alpha$, if $\tau[f_1]_\alpha = \tau[f_2]_\alpha = \mathsf{T}$.

b. $\mathcal{L}_{\text{partial}}[f_1]_\alpha \cup \mathcal{L}_{\text{partial}}[f_2]_\alpha$, if $\tau[f_1]_\alpha = \tau[f_2]_\alpha = \mathsf{F}$.

c. $\mathcal{L}_{\text{partial}}[f_2]_\alpha$, if $\tau[f_1]_\alpha = \mathsf{T}, \tau[f_2]_\alpha = \mathsf{F}$.

d. $\mathcal{L}_{\text{partial}}[f_1]_\alpha$, if $\tau[f_1]_\alpha = \mathsf{F}, \tau[f_2]_\alpha = \mathsf{T}$.

if $\mathsf{Affected}(f_1) = \mathsf{Affected}(f_2) = \mathsf{T}$.

Fig. 38. INFuse's partial link generation semantics for the `and` formula.

(i.e., reevaluating truth values and regenerating links) nodes in one sub-tree of node r per context change (let it be U), (4) the average number of sub-trees of node r per context change (let it be B). Since ECC conducts full checking (i.e., visiting every node three times) upon every single change, its time complexity of one single change is:

$$3 \cdot N \cdot B \tag{23}$$

$\mathcal{L}_{\text{partial}}[(f_1) \text{ or } (f_2)]_\alpha =$

(1) $\mathcal{L}_0[(f_1) \text{ or } (f_2)]_\alpha$, if $\mathsf{Affected}(f_1) = \mathsf{Affected}(f_2) = \mathsf{F}$.

(2) a. $\mathcal{L}_{\text{partial}}[f_1]_\alpha \cup \mathcal{L}_0[f_2]_\alpha$, if $\tau[f_1]_\alpha = \tau[f_2]_\alpha = \mathsf{T}$.

b. $\mathcal{L}_{\text{partial}}[f_1]_\alpha \otimes \mathcal{L}_0[f_2]_\alpha$, if $\tau[f_1]_\alpha = \tau[f_2]_\alpha = \mathsf{F}$.

c. $\mathcal{L}_{\text{partial}}[f_1]_\alpha$, if $\tau[f_1]_\alpha = \mathsf{T}, \tau[f_2]_\alpha = \mathsf{F}$.

d. $\mathcal{L}_0[f_2]_\alpha$, if $\tau[f_1]_\alpha = \mathsf{F}, \tau[f_2]_\alpha = \mathsf{T}$.

if $\mathsf{Affected}(f_1) = \mathsf{T}, \mathsf{Affected}(f_2) = \mathsf{F}$.

(3) a. $\mathcal{L}_0[f_1]_\alpha \cup \mathcal{L}_{\text{partial}}[f_2]_\alpha$, if $\tau[f_1]_\alpha = \tau[f_2]_\alpha = \mathsf{T}$.

b. $\mathcal{L}_0[f_1]_\alpha \otimes \mathcal{L}_{\text{partial}}[f_2]_\alpha$, if $\tau[f_1]_\alpha = \tau[f_2]_\alpha = \mathsf{F}$.

c. $\mathcal{L}_0[f_1]_\alpha$, if $\tau[f_1]_\alpha = \mathsf{T}, \tau[f_2]_\alpha = \mathsf{F}$.

d. $\mathcal{L}_{\text{partial}}[f_2]_\alpha$, if $\tau[f_1]_\alpha = \mathsf{F}, \tau[f_2]_\alpha = \mathsf{T}$.

if $\mathsf{Affected}(f_1) = \mathsf{F}, \mathsf{Affected}(f_2) = \mathsf{T}$.

(4) a. $\mathcal{L}_{\text{partial}}[f_1]_\alpha \cup \mathcal{L}_{\text{partial}}[f_2]_\alpha$, if $\tau[f_1]_\alpha = \tau[f_2]_\alpha = \mathsf{T}$.

b. $\mathcal{L}_{\text{partial}}[f_1]_\alpha \otimes \mathcal{L}_{\text{partial}}[f_2]_\alpha$, if $\tau[f_1]_\alpha = \tau[f_2]_\alpha = \mathsf{F}$.

c. $\mathcal{L}_{\text{partial}}[f_1]_\alpha$, if $\tau[f_1]_\alpha = \mathsf{T}, \tau[f_2]_\alpha = \mathsf{F}$.

d. $\mathcal{L}_{\text{partial}}[f_2]_\alpha$, if $\tau[f_1]_\alpha = \mathsf{F}, \tau[f_2]_\alpha = \mathsf{T}$.

if $\mathsf{Affected}(f_1) = \mathsf{Affected}(f_2) = \mathsf{T}$.

Fig. 39. INFuse's partial link generation semantics for the `or` formula.

ConC spreads the complexity into B threads, thus, its time complexity of one single change is:

$$3 \cdot N \tag{24}$$

For PCC, it has to consider two cases. On the one hand, if the context change is an addition change, PCC would visit new added nodes (i.e., the number is Δ) three times and updated nodes (i.e., the number is U) twice. Typically, adding new nodes dominates the time complexity. Therefore, its time complexity of one single change is:

$$3 \cdot \Delta \cdot B \tag{25}$$

On the other hand, if the context change is a deletion change, PCC would remove Δ nodes and update U nodes. Therefore, its time complexity of one single change is:

$$(1 \cdot \Delta + 2 \cdot U) \cdot B \tag{26}$$

In the following, we estimate the time complexity of existing checking techniques in concrete cases based on the above analysis.

Parallel structure. Based on Eq. (9) in the main article, initially, the number of nodes in one sub-tree of node r is $O(e\sum_{i=1}^{k-1} h_i) = O(\frac{n_0}{e})$.

(1) *Only ASet changes.* In this case, the number of nodes in one sub-tree of node r changes from $O(\frac{n_0}{e})$ to $O(\frac{e+a}{e} \cdot \frac{n_0}{e})$ evenly. Therefore, the average number of that (a.k.a, N) is $O(\frac{2e+a}{2e} \cdot \frac{n_0}{e})$, and the number of newly added nodes is $O(\frac{a}{e} \cdot \frac{n_0}{e})$. Since $ASet$ is regarded as a list containing $O(ka)$ addition changes, the average number of newly added nodes in one sub-tree per change (a.k.a, Δ) is $O(\frac{1}{ke} \cdot \frac{n_0}{e})$ Similarly, the number of sub-tree of node r changes from $O(e)$ to $O(e+a)$, thus, the average number of that (a.k.a, B) is $O(\frac{2e+a}{2})$. Based on the time complexity analysis of INFuse in our main article, if every context is affected by

$$\begin{aligned}
&\mathcal{L}_{\text{partial}}[(f_1)\text{ implies }(f_2)]_\alpha = \\
&\quad (1)\ \mathcal{L}_0[(f_1)\text{ implies }(f_2)]_\alpha,\ \text{if affected}(f_1) = \text{affected}(f_2) = \mathsf{F}. \\
&\quad (2)\ \text{a. FlipSet}(\mathcal{L}_{\text{partial}}[f_1]_\alpha) \otimes \mathcal{L}_0[f_2]_\alpha,\ \text{if } \tau[f_1]_\alpha = \mathsf{T}, \tau[f_2]_\alpha = \mathsf{F}. \\
&\quad\quad \text{b. FlipSet}(\mathcal{L}_{\text{partial}}[f_1]_\alpha) \cup \mathcal{L}_0[f_2]_\alpha,\ \text{if } \tau[f_1]_\alpha = \mathsf{F}, \tau[f_2]_\alpha = \mathsf{T}. \\
&\quad\quad \text{c. } \mathcal{L}_0[f_2]_\alpha,\ \text{if } \tau[f_1]_\alpha = \tau[f_2]_\alpha = \mathsf{T}. \\
&\quad\quad \text{d. FlipSet}(\mathcal{L}_{\text{partial}}[f_1]_\alpha),\ \text{if } \tau[f_1]_\alpha = \tau[f_2]_\alpha = \mathsf{F}. \\
&\quad \text{if Affected}(f_1) = \mathsf{T}, \text{Affected}(f_2) = \mathsf{F}. \\
&\quad (3)\ \text{a. FlipSet}(\mathcal{L}_0[f_1]_\alpha) \otimes \mathcal{L}_{\text{partial}}[f_2]_\alpha,\ \text{if } \tau[f_1]_\alpha = \mathsf{T}, \tau[f_2]_\alpha = \mathsf{F}. \\
&\quad\quad \text{b. FlipSet}(\mathcal{L}_0[f_1]_\alpha) \cup \mathcal{L}_{\text{partial}}[f_2]_\alpha,\ \text{if } \tau[f_1]_\alpha = \mathsf{F}, \tau[f_2]_\alpha = \mathsf{T}. \\
&\quad\quad \text{c. } \mathcal{L}_{\text{partial}}[f_2]_\alpha,\ \text{if } \tau[f_1]_\alpha = \tau[f_2]_\alpha = \mathsf{T}. \\
&\quad\quad \text{d. FlipSet}(\mathcal{L}_0[f_1]_\alpha),\ \text{if } \tau[f_1]_\alpha = \tau[f_2]_\alpha = \mathsf{F}. \\
&\quad \text{if Affected}(f_1) = \mathsf{F}, \text{Affected}(f_2) = \mathsf{T}. \\
&\quad (4)\ \text{a. FlipSet}(\mathcal{L}_{\text{partial}}[f_1]_\alpha) \otimes \mathcal{L}_{\text{partial}}[f_2]_\alpha, \text{if } \tau[f_1]_\alpha = \mathsf{T}, \tau[f_2]_\alpha = \mathsf{F}. \\
&\quad\quad \text{b. FlipSet}(\mathcal{L}_{\text{partial}}[f_1]_\alpha) \cup \mathcal{L}_{\text{partial}}[f_2]_\alpha, \text{if } \tau[f_1]_\alpha = \mathsf{F}, \tau[f_2]_\alpha = \mathsf{T}. \\
&\quad\quad \text{c. } \mathcal{L}_{\text{partial}}[f_2]_\alpha,\ \text{if } \tau[f_1]_\alpha = \tau[f_2]_\alpha = \mathsf{T}. \\
&\quad\quad \text{d. FlipSet}(\mathcal{L}_{\text{partial}}[f_1]_\alpha),\ \text{if } \tau[f_1]_\alpha = \tau[f_2]_\alpha = \mathsf{F}. \\
&\quad \text{if Affected}(f_1) = \text{Affected}(f_2) = \mathsf{T}.
\end{aligned}$$

Fig. 40. INFUSE's partial link generation semantics for the `implies` formula.

$$\mathcal{L}_{\text{entire}}[\text{not } (f)]_\alpha = \text{FlipSet}(\mathcal{L}_{\text{entire}}[f]_\alpha).$$

$$\mathcal{L}_{\text{entire}}[bfunc(\gamma_1, \cdots, \gamma_n)]_\alpha = \emptyset.$$

Fig. 41. INFUSE's entire and partial link generation semantics for `not` and $bfunc$ formulas.

$$\begin{aligned}
&\mathcal{L}_{\text{partial}}[\text{not } (f)]_\alpha = \\
&\quad (1)\ \mathcal{L}_0[\text{not } (f)]_\alpha,\ \text{if Affected}(f) = \mathsf{F}. \\
&\quad (2)\ \text{FlipSet}(\mathcal{L}_{\text{partial}}[f]_\alpha),\ \text{if Affected}(f) = \mathsf{T}.
\end{aligned}$$

$$\mathcal{L}_{\text{partial}}[bfunc(\gamma_1, \cdots, \gamma_n)]_\alpha = \emptyset.$$

Fig. 42. INFUSE's partial link generation semantics for `not` and $bfunc$ formulas.

one context change respectively, the average number of updated nodes is $O(\frac{1}{2}\sum_{i=1}^{k-1}(h-h_i))$. Since there are k contexts in total, the average number of updated nodes per context change (a.k.a, U) is $O(\frac{1}{2k}\sum_{i=1}^{k-1}(h-h_i))$. Therefore, we can estimate time complexities of existing techniques for one single context change as follows:

$$ECC_{single} = 3 \cdot O(\frac{2e+a}{2e} \cdot \frac{n_0}{e}) \cdot O(\frac{2e+a}{2}) = O(\frac{3(2e+a)^2}{4e} \cdot \frac{n_0}{e}) \quad (27)$$

$$ConC_{single} = 3 \cdot O(\frac{2e+a}{2e} \cdot \frac{n_0}{e}) = O(\frac{3(2e+a)}{2e} \cdot \frac{n_0}{e}) \quad (28)$$

$$PCC_{single} = 3 \cdot O(\frac{1}{ke} \cdot \frac{n_0}{e}) \cdot O(\frac{2e+a}{2}) = O(\frac{3(2e+a)}{2ke} \cdot \frac{n_0}{e}) \quad (29)$$

Since there are $O(ka)$ context changes in total, the overall time complexity of existing techniques are as follows:

$$ECC_{overall} = O(\frac{3ka(2e+a)^2}{4e} \cdot \frac{n_0}{e}) \quad (30)$$

$$ConC_{overall} = O(\frac{3ka(2e+a)}{2e} \cdot \frac{n_0}{e}) \quad (31)$$

$$PCC_{overall} = O(\frac{3a(2e+a)}{2e} \cdot \frac{n_0}{e}) \quad (32)$$

(2) *Only DSet changes.* The only difference between this case and the $ASet$ case is that some nodes are removed instead of newly added. Therefore, we can similarly obtain the following expressions: $N = O(\frac{2e-d}{2e} \cdot \frac{n_0}{e})$, $\Delta = O(\frac{1}{ek} \cdot \frac{n_0}{e})$, $B = O(\frac{2e-d}{2})$, and $U = O(\frac{1}{2ke} \cdot \frac{n_0}{e})$. Consequently, the overall time complexity of existing techniques are as follows:

$$ECC_{overall} = O(\frac{3kd(2e-d)^2}{4e} \cdot \frac{n_0}{e}) \quad (33)$$

$$ConC_{overall} = O(\frac{3kd(2e-d)}{2e} \cdot \frac{n_0}{e}) \quad (34)$$

$$PCC_{overall} = O(\frac{d(2e-d)}{e} \cdot \frac{n_0}{e}) \quad (35)$$

(3) *Only USet changes.* Since $USet$ is regarded as a list containing $O(ku)$ deletion changes and then $O(ku)$ addition changes, we can consider this case as a $DSet$ case and its reverse. Therefore, by adapting the analysis from the $DSet$ case, we can estimate the overall time complexity of existing techniques in this case as follows:

$$ECC_{overall} = O(\frac{3ku(2e-u)^2}{2e} \cdot \frac{n_0}{e}) \quad (36)$$

$$ConC_{overall} = O(\frac{3ku(2e-u)}{e} \cdot \frac{n_0}{e}) \quad (37)$$

$$PCC_{overall} = O(\frac{3u(2e-u)}{e} \cdot \frac{n_0}{e}) \quad (38)$$

Nested structure. Based on Eq. (15) in the main article, the number of nodes in one sub-tree of node r is $O(\frac{1}{2}he^{k-1}) = O(\frac{n_0}{e})$.

(1) *Only ASet changes.* In this case, the number of nodes in one sub-tree of node r changes from $O(\frac{n_0}{e})$ to $O((\frac{e+a}{e})^{k-1}\frac{n_0}{e})$ evenly. Therefore, $N = O(\frac{1}{2}(1+(\frac{e+a}{e})^{k-1})\frac{n_0}{e})$. Since the exponent expression grows rapidly, $(\frac{e+a}{e})^{k-1}$ is supposed to be much greater

than 1. Therefore, we roughly estimate N as $O(\frac{1}{2}(\frac{e+a}{e})^{k-1}\frac{n_0}{e})$. Similarly, we can estimate Δ as $O(\frac{1}{ka}(\frac{e+a}{e})^{k-1}\frac{n_0}{e})$ and U as $O(\frac{1}{2ke}(\frac{e+a}{e})^{k-2}\frac{n_0}{e})$. B is $O(\frac{2e+a}{2})$, which is the same to that in $ASet$ case on parallel structure. Therefore, we can estimate the overall time complexity of existing techniques as follows:

$$ECC_{overall} = O(\frac{3ka(2e+a)}{4}(\frac{e+a}{e})^{k-1}\cdot\frac{n_0}{e}) \tag{39}$$

$$ConC_{overall} = O(\frac{3ka}{2}(\frac{e+a}{e})^{k-1}\cdot\frac{n_0}{e}) \tag{40}$$

$$PCC_{overall} = O(\frac{3(2e+a)}{2}(\frac{e+a}{e})^{k-1}\cdot\frac{n_0}{e}) \tag{41}$$

(2) *Only $DSet$ changes.* In this case, the number of nodes in one subtree of node r decreases from $O(\frac{n_0}{e})$ to $O((\frac{e-d}{e})^{k-1}\frac{n_0}{e})$. Since $\frac{e-d}{e} < 1$ and exponent expression changes rapidly, $(\frac{e-d}{e})^{k-1}$ is supposed to be much less than 1. Therefore, we can similarly obtain the following expressions: $N = O(\frac{1}{2}\cdot\frac{n_0}{e})$, $\Delta = O(\frac{1}{kd}\cdot\frac{n_0}{e})$, $B = O(\frac{2e-d}{2})$, and $U = O(\frac{1}{2ke}\cdot\frac{n_0}{e})$. Then the overall time complexity of existing techniques are as follows:

$$ECC_{overall} = O(\frac{3kd(2e-d)}{4}\cdot\frac{n_0}{e}) \tag{42}$$

$$ConC_{overall} = O(\frac{3kd}{2}\cdot\frac{n_0}{e}) \tag{43}$$

$$PCC_{overall} = O(\frac{(e+d)(2e-d)}{2e}\cdot\frac{n_0}{e}) \tag{44}$$

(3) *Only $USet$ changes.* Considering this case as a $DSet$ case and its reverse, we can adapt the analysis from the $DSet$ case and estimate the overall time complexity of existing techniques as follows:

$$ECC_{overall} = O(\frac{3ku(2e-u)}{2}\cdot\frac{n_0}{e}) \tag{45}$$

$$ConC_{overall} = O(3ku\cdot\frac{n_0}{e}) \tag{46}$$

$$PCC_{overall} = O(\frac{4e^2-u^2}{e}\cdot\frac{n_0}{e}) \tag{47}$$

References

Bashir, R.S., Lee, S.P., ur Rehman Khan, S., Chang, V., Farid, S., 2016. UML models consistency management: Guidelines for software quality manager. Int. J. Inf. Manag. 36 (6), 883–899. http://dx.doi.org/10.1016/j.ijinfomgt.2016.05.024.

Brun, Y., Holmes, R., Ernst, M.D., Notkin, D., 2011. Proactive detection of collaboration conflicts. In: Gyimóthy, T., Zeller, A. (Eds.), SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13). Szeged, Hungary, September 5-9, 2011, ACM, pp. 168–178. http://dx.doi.org/10.1145/2025113.2025139.

Bu, Y., Gu, T., Tao, X., Li, J., Chen, S., Lu, J., 2006. Managing quality of context in pervasive computing. In: Sixth International Conference on Quality Software. QSIC 2006, 26-28 October 2006, Beijing, China, IEEE Computer Society, pp. 193–200. http://dx.doi.org/10.1109/QSIC.2006.38.

Capra, L., Emmerich, W., Mascolo, C., 2003. CARISMA: Context-aware reflective middleware system for mobile applications. IEEE Trans. Softw. Eng. 29 (10), 929–945. http://dx.doi.org/10.1109/TSE.2003.1237173.

Chaudhuri, S., Ganjam, K., Ganti, V., Motwani, R., 2003. Robust and efficient fuzzy match for online data cleaning. In: Ives, Z., Papakonstantinou, Y., Halevy, A. (Eds.), Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data. San Diego, California, USA, June 9-12, 2003, ACM, pp. 313–324. http://dx.doi.org/10.1145/872757.872796.

Chen, J., Qin, Y., Wang, H., Xu, C., 2022a. Simulation might change your results: A comparison of context-aware system input validation in simulated and physical environments. J. Comput. Sci. Technol. 37 (1), 83–105. http://dx.doi.org/10.1007/s11390-021-1669-1.

Chen, C., Wang, H., Zhang, L., Xu, C., Yu, P., 2022b. Minimizing link generation in constraint checking for context inconsistency detection. In: Pastore, F., Zhang, L. (Eds.), 2022 IEEE International Symposium on Software Reliability Engineering. ISSRE 2022, Charlotte, North Carolina, USA, Oct-Nov 2022, IEEE, pp. 13–24.

Chen, C., Ye, C., Jacobsen, H., 2011. Hybrid context inconsistency resolution for context-aware services. In: Cook, D., Indulska, J. (Eds.), Ninth Annual IEEE International Conference on Pervasive Computing and Communications, PerCom 2011, 21-25 March 2011, Seattle, WA, USA, Proceedings. IEEE, pp. 10–19. http://dx.doi.org/10.1109/PERCOM.2011.5767574.

Chomicki, J., Lobo, J., Naqvi, S.A., 2003. Conflict resolution using logic programming. IEEE Trans. Knowl. Data Eng. 15 (1), 244–249. http://dx.doi.org/10.1109/TKDE.2003.1161596.

Chun, C., Jeon, K.M., Kim, T., Choi, W., 2019. Drone noise reduction using deep convolutional autoencoder for UAV acoustic sensor networks. In: Abdelzaher, T., Wang, X., Demirkol, I. (Eds.), 16th IEEE International Conference on Mobile Ad Hoc and Sensor Systems Workshops. MASS Workshops 2019, Monterey, CA, USA, November 4-7, 2019, IEEE, pp. 168–169. http://dx.doi.org/10.1109/MASSW.2019.00043.

Darcy, P., Stantic, B., Sattar, A., 2011. An intelligent approach to handle False-Positive Radio Frequency Identification Anomalies. Intell. Data Anal. 15 (6), 931–954. http://dx.doi.org/10.3233/IDA-2011-0503.

Davies, A., 2017. The numbers don't lie: Self-driving cars are getting good. https://www.wired.com/2017/02/california-dmv-autonomous-car-disengagement/.

Demuth, A., Riedl-Ehrenleitner, M., Egyed, A., 2016. Efficient detection of inconsistencies in a multi-developer engineering environment. In: Lo, D., Apel, S., Khurshid, S. (Eds.), Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. ASE 2016, Singapore, September 3-7, 2016, ACM, pp. 590–601. http://dx.doi.org/10.1145/2970276.2970304.

Fescioglu-Ünver, N., Choi, S.H., Sheen, D., Kumara, S.R.T., 2015. RFID in production and service systems: Technology, applications and issues. Inf. Syst. Front. 17 (6), 1369–1380. http://dx.doi.org/10.1007/s10796-014-9518-1.

Guo, B., Wang, H., Xu, C., Lu, J., 2017. GEAS: generic adaptive scheduling for high-efficiency context inconsistency detection. In: Mei, H., Zhang, L., Zimmermann, T. (Eds.), 2017 IEEE International Conference on Software Maintenance and Evolution. ICSME 2017, Shanghai, China, September 17-22, 2017, IEEE Computer Society, pp. 137–147. http://dx.doi.org/10.1109/ICSME.2017.10.

Handley, H.A.H., Khallouli, W., Huang, J., Edmonson, W., Kibret, N., 2021. Maintaining the consistency of sysml model exports to XML metadata interchange (XMI). In: Rassa, B., Givigi, S. (Eds.), IEEE International Systems Conference. SysCon 2021, Vancouver, BC, Canada, April 15 - May 15, 2021, IEEE, pp. 1–8. http://dx.doi.org/10.1109/SysCon48628.2021.9447105.

Jeffery, S.R., Garofalakis, M.N., Franklin, M.J., 2006. Adaptive cleaning for RFID data streams. In: Dayal, U., Whang, K., Lomet, D.B., Alonso, G., Lohman, G.M., Kersten, M.L., Cha, S.K., Kim, Y. (Eds.), Proceedings of the 32nd International Conference on Very Large Data Bases. Seoul, Korea, September 12-15, 2006, ACM, pp. 163–174, URL: http://dl.acm.org/citation.cfm?id=1164143.

Jongeling, R., Ciccozzi, F., Cicchetti, A., Carlson, J., 2019. Lightweight consistency checking for agile model-based development in practice. J. Object Technol. 18 (2), 11:1–20. http://dx.doi.org/10.5381/jot.2019.18.2.a11.

Kehrer, T., Kelter, U., Taentzer, G., 2013. Consistency-preserving edit scripts in model versioning. In: Denney, E., Bultan, T., Zeller, A. (Eds.), 2013 28th IEEE/ACM International Conference on Automated Software Engineering. ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013, IEEE, pp. 191–201. http://dx.doi.org/10.1109/ASE.2013.6693079.

Khelladi, D.E., Kretschmer, R., Egyed, A., 2019. Detecting and exploring side effects when repairing model inconsistencies. In: Nierstrasz, O., Gray, J., d. S. Oliveira, B.C. (Eds.), Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering. SLE 2019, Athens, Greece, October 20-22, 2019, ACM, pp. 113–126. http://dx.doi.org/10.1145/3357766.3359546.

Kuo, P., Lin, S., Hu, J., Huang, C., 2021. Multi-sensor context-aware based chatbot model: An application of humanoid companion robot. Sensors 21 (15), 5132. http://dx.doi.org/10.3390/s21155132.

Lahmeri, M., Kishk, M.A., Alouini, M., 2022. Charging techniques for UAV-assisted data collection: Is laser power beaming the answer? IEEE Commun. Mag. 60 (5), 50–56. http://dx.doi.org/10.1109/MCOM.001.2100871.

Le, H.A., Dao, T., Truong, N., 2017. A formal approach to checking consistency in software refactoring. Mob. Netw. Appl. 22 (2), 356–366. http://dx.doi.org/10.1007/s11036-017-0807-z.

Li, Y., Wang, S., Nguyen, T.N., 2021. A context-based automated approach for method name consistency checking and suggestion. In: Juristo, N., van Deursen, A., Xie, T. (Eds.), 43rd IEEE/ACM International Conference on Software Engineering. ICSE 2021, Madrid, Spain, 22-30 May 2021, IEEE, pp. 574–586. http://dx.doi.org/10.1109/ICSE43902.2021.00060.

Limón, Y., Bárcenas, E., Benítez-Guerrero, E., Gomez, J., 2019. Consistency checking of attention aware systems. In: Galindo, M.J.O., Marcial-Romero, J.R., Cortés, C.Z., Parra, P.P. (Eds.), Proceedings of the Twelfth Latin American Workshop on Logic/Languages, Algorithms and New Methods of Reasoning. Puebla, Mexico, November 15, 2019, In: CEUR Workshop Proceedings, vol. 2585, CEUR-WS.org, pp. 13–23, URL: http://ceur-ws.org/Vol-2585/paper2.pdf.

Limón, Y., Bárcenas, E., Benítez-Guerrero, E., Molero, G., 2018. On the consistency of context-aware systems. J. Intell. Fuzzy Systems 34 (5), 3373–3383. http://dx.doi.org/10.3233/JIFS-169518.

Mayr-Dorn, C., Kretschmer, R., Egyed, A., Heradio, R., Fernández-Amorós, D., 2021a. Inconsistency-tolerating guidance for software engineering processes. In: Juristo, N., Lago, P., Murphy, G. (Eds.), 43rd IEEE/ACM International Conference on Software Engineering: New Ideas and Emerging Results. ICSE NIER 2021, Madrid, Spain, May 25-28, 2021, IEEE, pp. 6–10. http://dx.doi.org/10.1109/ICSE-NIER52604.2021.00010.

Mayr-Dorn, C., Vierhauser, M., Bichler, S., Keplinger, F., Cleland-Huang, J., Egyed, A., Mehofer, T., 2021b. Supporting quality assurance with automated process-centric quality constraints checking. In: Juristo, N., van Deursen, A., Xie, T. (Eds.), 43rd IEEE/ACM International Conference on Software Engineering. ICSE 2021, Madrid, Spain, 22-30 May 2021, IEEE, pp. 1298–1310. http://dx.doi.org/10.1109/ICSE43902.2021.00118.

Mazumdar, N., Roy, S., Nag, A., Singh, J.P., 2022. A buffer-aware dynamic UAV trajectory design for data collection in resource-constrained IoT frameworks. Comput. Electr. Eng. 100, 107934. http://dx.doi.org/10.1016/j.compeleceng.2022.107934.

Messaoudi, N., Chaoui, A., Bettaz, M., 2017. An approach to UML consistency checking based on compositional semantics. Int. J. Embed. Real Time Commun. Syst. 8 (2), 1–23. http://dx.doi.org/10.4018/IJERTCS.2017070101.

Murphy, A.L., Picco, G.P., Roman, G., 2006. LIME: a coordination model and middleware supporting mobility of hosts and agents. ACM Trans. Softw. Eng. Methodol. 15 (3), 279–328. http://dx.doi.org/10.1145/1151695.1151698.

Nentwich, C., Capra, L., Emmerich, W., Finkelstein, A., 2002. Xlinkit: a consistency checking and smart link generation service. ACM Trans. Internet Tech. 2 (2), 151–185. http://dx.doi.org/10.1145/514183.514186.

Patil, K., Bansal, V., Dhateria, V., Narayankhedkar, S., 2015. Probable causes of RFID tag read unreliability in supermarkets and proposed solutions. In: Zhang, T., El-Maleh, K., Wang, H., lav Kisacanin, B. (Eds.), International Conference on Information Processing. IEEE, pp. 392–397. http://dx.doi.org/10.1109/INFOP.2015.7489414.

pollen Sense, 2022. Pollen Wise - What's in your air, when and where. https://play.google.com/store/apps/details?id=com.PollenSense.PollenWise.

Rao, J., Doraiswamy, S., Thakkar, H., Colby, L.S., 2006. A deferred cleansing method for RFID data analytics. In: Dayal, U., Whang, K., Lomet, D.B., Alonso, G., Lohman, G.M., Kersten, M.L., Cha, S.K., Kim, Y. (Eds.), Proceedings of the 32nd International Conference on Very Large Data Bases. Seoul, Korea, September 12-15, 2006, ACM, pp. 175–186, URL: http://dl.acm.org/citation.cfm?id=1164144.

Reiss, S.P., 2006. Incremental maintenance of software artifacts. IEEE Trans. Softw. Eng. 32 (9), 682–697. http://dx.doi.org/10.1109/TSE.2006.91.

Tan, Z., Nguyen, A.H.T., Khong, A.W.H., 2019. An efficient dilated convolutional neural network for UAV noise reduction at low input SNR. In: Zheng, T.F., Yu, H., Dang, J., Siu, W., Kiya, H. (Eds.), 2019 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference. APSIPA ASC 2019, Lanzhou, China, November 18-21, 2019, pp. 1885–1892. http://dx.doi.org/10.1109/APSIPAASC47483.2019.9023324.

Wang, H., Xu, C., Guo, B., Ma, X., Lu, J., 2021. Generic adaptive scheduling for efficient context inconsistency detection. IEEE Trans. Softw. Eng. 47 (3), 464–497. http://dx.doi.org/10.1109/TSE.2019.2898976.

Waymo, 2022. Waymo. https://waymo.com.

Wei, B., Sun, J., 2021. Leveraging SPARQL queries for UML consistency checking. Int. J. Softw. Eng. Knowl. Eng. 31 (4), 635–654. http://dx.doi.org/10.1142/S0218194021500170.

Wei, Y., Xue, M., Liu, X., Xu, P., 2022. Data fusing and joint training for learning with noisy labels. Frontiers of Computer Science 16 (6), 166338. http://dx.doi.org/10.1007/s11704-021-1208-9, https://journal.hep.com.cn/fcs/EN/abstract/article_30329.shtml.

Xu, C., Cheung, S., Chan, W.K., 2007a. Goal-directed context validation for adaptive ubiquitous systems. In: Cheng, B.H., de Lemos, R., Fickas, S., Garlan, D., Litoiu, M., Magee, J., Müller, H.A., Taylor, R.N. (Eds.), 2007 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems. SEAMS 2007, Minneapolis Minnesota, USA, May 20-26, 2007, IEEE Computer Society, p. 17. http://dx.doi.org/10.1109/SEAMS.2007.8.

Xu, C., Cheung, S.C., Chan, W.K., Ye, C., 2007b. On impact-oriented automatic resolution of pervasive context inconsistency. In: Crnkovic, I., Bertolino, A. (Eds.), Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007. Dubrovnik, Croatia, September 3-7, 2007, ACM, pp. 569–572. http://dx.doi.org/10.1145/1287624.1287712.

Xu, C., Cheung, S., Chan, W.K., Ye, C., 2008. Heuristics-based strategies for resolving context inconsistencies in pervasive computing applications. In: 28th IEEE International Conference on Distributed Computing Systems. ICDCS 2008, 17-20 June 2008, Beijing, China, IEEE Computer Society, pp. 713–721. http://dx.doi.org/10.1109/ICDCS.2008.46.

Xu, C., Cheung, S.C., Chan, W.K., Ye, C., 2010. Partial constraint checking for context consistency in pervasive computing. ACM Trans. Softw. Eng. Methodol. 19 (3), 9:1–9:61. http://dx.doi.org/10.1145/1656250.1656253.

Xu, C., Cheung, S., Lo, C., Leung, K., Wei, J., 2004. Cabot: On the ontology for the middleware support of context-aware pervasive applications. In: Jin, H., Gao, G.R., Xu, Z., Chen, H. (Eds.), Network and Parallel Computing, IFIP International Conference, NPC 2004, Wuhan, China, October 18-20, 2004, Proceedings. In: Lecture Notes in Computer Science, vol. x3222, Springer, pp. 568–575. http://dx.doi.org/10.1007/978-3-540-30141-7_85.

Xu, C., Cheung, S.C., Ma, X., Cao, C., Lu, J., 2012. Adam: Identifying defects in context-aware adaptation. J. Syst. Softw. 85 (12), 2812–2828. http://dx.doi.org/10.1016/j.jss.2012.04.078.

Xu, C., Liu, Y., Cheung, S.C., Cao, C., Lv, J., 2013. Towards context consistency by concurrent checking for internetware applications. Sci. China Inf. Sci. 56 (8), 1–20. http://dx.doi.org/10.1007/s11432-013-4907-5.

Xu, C., Ma, X., Cao, C., Lu, J., 2011. Minimizing the side effect of context inconsistency resolution for ubiquitous computing. In: Puiatti, A., Gu, T. (Eds.), Mobile and Ubiquitous Systems: Computing, Networking, and Services - 8th International ICST Conference. MobiQuitous 2011, Copenhagen, Denmark, December 6-9, 2011, Revised Selected Papers, In: Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol. 104, Springer, pp. 285–297. http://dx.doi.org/10.1007/978-3-642-30973-1_29.

Xu, C., Qin, Y., Yu, P., Cao, C., Lu, J., 2020. Theories and techniques for growing software: Paradigm and beyond. Sci. Sin. Inf. 50 (11), 1595–1611.

Xu, C., Xi, W., Cheung, S., Ma, X., Cao, C., Lu, J., 2015. Cina: Suppressing the detection of unstable context inconsistency. IEEE Trans. Softw. Eng. 41 (9), 842–865. http://dx.doi.org/10.1109/TSE.2015.2418760.

Yoon, I., Noh, D.K., 2022. Adaptive data collection using UAV with wireless power transfer for wireless rechargeable sensor networks. IEEE Access 10, 9729–9743. http://dx.doi.org/10.1109/ACCESS.2022.3144846.

Zhang, L., Wang, H., Xu, C., Yu, P., 2022. INFUSE: towards efficient context consistency by incremental-concurrent check fusion. In: Avgeriou, P., Binkley, D. (Eds.), 2022 IEEE International Conference on Software Maintenance and Evolution. ICSME 2022, Limassol, Cyprus, October, 2022, IEEE, pp. 187–198.

Lingyu Zhang is a Ph.D. student with the Department of Computer Science and Technology at Nanjing University, China. He received his B.Sc. degree in computer science and technology from Nanjing University in 2021. His research interests include software consistency management, software testing and analysis, and software methodologies.

Huiyan Wang received her doctoral degree in computer science and engineering from Nanjing University, China. She is now an assistant researcher with the Department of Computer Science and Technology at Nanjing University, China. Her research interests include intelligent software quality assurance, context management, software analyses and testing.

Chuayang Chen is a M.Sc. student with the Department of Computer Science and Technology at Nanjing University, China. He received his B.Sc. degree in 2020 and began his M.Sc. program under the supervision of assistant researcher Huiyan Wang and professor Chang Xu.

Chang Xu received his doctoral degree in computer science and engineering from The Hong Kong University of Science and Technology, China. He is now a full professor with the Department of Computer Science and Technology at Nanjing University, China. His research interests include big data software engineering, intelligent software testing and analysis, and adaptive and autonomous software systems.

Ping Yu received her doctoral degree in Computer Science and Technology in 2008 from Nanjing University. She is an associate professor with the State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology at Nanjing University. Her research interests include intelligent software engineering, cloud computing and big data technology.