INFUSE: Towards Efficient Context Consistency by Incremental-Concurrent Check Fusion

Lingyu Zhang[†‡], Huiyan Wang[*†‡], Chang Xu[*†‡], and Ping Yu[†‡]
[†]State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
[‡]Department of Computer Science and Technology, Nanjing University, Nanjing, China
zly@smail.nju.edu.cn, {why, changxu, yuping}@nju.edu.cn

***Abstract*—Nowadays applications are getting increasingly attractive by being capable of adapting their behaviors based on their understanding to running environments (a.k.a. contexts). However, such capability can be subject to illness or even unexpected crash, when contexts, for suffering environmental noises, become inaccurate or even conflict with each other. Fortunately, various constraint checking techniques have been proposed to validate contexts against consistency constraints, in order to guard context consistency for applications in a timely manner. However, with the growth of environmental dynamics and context volume, it is getting more and more challenging to check context consistency in time. In this paper, we propose a novel approach, INFUSE, to soundly fuse together two lines of techniques, namely, incremental checking and concurrent checking, for 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 the challenges by automatically deciding maximal concurrent boundaries for context changes under checking (i.e., what-correctness problem), and soundly fusing incremental and concurrent checking for context consistency (i.e., how-correctness problem), with theoretical guarantees. Our experimental evaluation with real-world data shows that INFUSE could improve constraint checking efficiency by 18.6x–171.1x, as compared with existing state-of-the-art techniques.**

***Index Terms*—*Technological, constraint checking, context consistency, check fusion.**

I. INTRODUCTION

Consistency management for software artifacts (e.g., edit script [1], UML models [2]–[4], and XML documents [5]–[7]) has received extensive research attention [8]. In the recent decades, there is an increasing demand for managing the consistency of *contexts*, in order to support reliable adaptation behaviors in self-adaptive or context-aware applications. Unlike traditional software artifacts, contexts, representing an application's understanding to its running environment, are prone to frequent changes, and thus call for efficient constraint checking techniques for their runtime validation.

The validation is conducted by checking the contexts collected by an application against a set of *consistency constraints* [5], [9] and any constraint violation indicates the detection of a *context inconsistency*. Various constraint checking techniques [5], [9]–[12] have been studied with different efficiency benefits and costs, e.g., xlinkit [5], working in a full checking way as the correctness baseline, PCC [10], checking incrementally by reusing previous results, and Con-C [11], checking concurrently basic units that carry similar workloads. However, with the growth of environmental dynamics and context volume, it is getting increasingly challenging to validate context consistency in a timely manner, causing missed inconsistencies or wrong reports [9].

An intuition is to fuse incremental checking (e.g., PCC [10]) and concurrent checking (e.g., Con-C [11]) for even higher efficiency. Indeed, they were developed from two orthogonal dimensions, but their fusion is non-trivial, with no substantial progress after nearly one decade since their initial proposal. The challenge probably comes from the following 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 micro parts (i.e., larger parts not easy for analysis), but the latter requires macro arrangements (i.e., smaller parts not useful 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). On the other hand, if one aggressively enlarges the analysis granularity of incremental checking, improper grouping of context changes as a whole can lead to wrong results, denying the purpose of more efficient checking.

In this paper, we propose INFUSE, (Incremental-CoNcurrent Fusion ChEcking), to address the two challenges from the above gap: (1) *What-correctness* problem: automatically analyzing and deciding the boundaries of collected context changes 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: soundly switching between incremental and concurrent checking upon the context changes grouped as a whole for higher efficiency. Both challenges are addressed with theoretical guarantees.

We experimentally evaluated INFUSE and compared it to existing constraint checking techniques on dynamic application scenarios with real-world data following existing work [9]–[12]. The experimental results show that INFUSE could dramatically boost the checking efficiency (up to 18.6x, 105.4x, and 171.1x improvements for light-, median-, and

[*]Corresponding authors.

heavy-workload scenarios, respectively), as compared to existing techniques. When put into the practical scenario simulation, INFUSE won with high efficiency and 100% correctness of checking results, while existing techniques caused up to 98.7% false negatives and 96.0% false positives.

In summary, we make the following contributions:

- We proposed a novel constraint checking approach, INFUSE, with incremental-concurrent checking techniques properly fused.
- We proved 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 evaluated INFUSE and compared it to state-of-the-art techniques, observing substantial efficiency improvement and desirable checking correctness.

The remainder of this article is organized as follows: Section II introduces the background and formulates our problem. Section III elaborates on INFUSE's methodology with details. Section IV evaluates INFUSE with real-world scenarios. Section V discusses the related work in recent years, and finally Section VI concludes this paper.

II. BACKGROUND

A. Preliminary

We define a *context* as a piece of information about an application's running environment (e.g., location, user, activity, etc.) [9], [10], [12]. Each context can be modeled as a finite set of relevant elements. For example, in a package delivery application [9], [10] that schedules transportation robots across warehouse, all robots currently in warehouse x can be modeled by a context $C_{\mathsf{x}} = \{\mathsf{r}_1, \mathsf{r}_2, \cdots\}$, 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 $C_{\mathsf{x}} = \{\mathsf{r}_1, \mathsf{r}_2\}$. If robot r_3 enters or r_2 leaves the warehouse, we have context changes $<+, C_{\mathsf{x}}, \mathsf{r}_3>$ or $<-, C_{\mathsf{x}}, \mathsf{r}_2>$.

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

To validate contexts, one could define *consistency constraints* [5], [9], which model physical laws or application-specific requirements [5], [9], [10], and check whether any constraint is violated (when yes, an *inconsistency* is detected). Existing work [9], [10], [12] has mostly followed a first order logic (FOL) styled language to specify consistency constraints:

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

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" [9], for the aforementioned application:

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

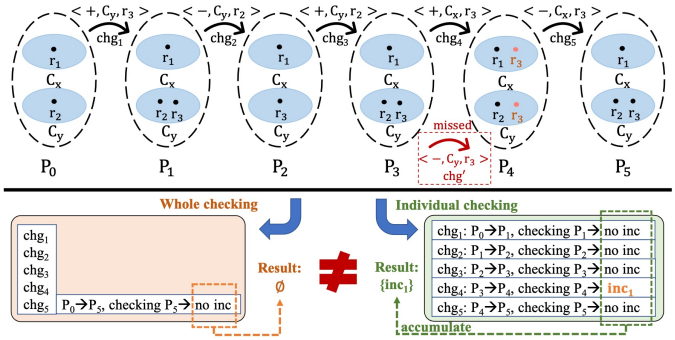


Fig. 1. An illustrative example (P_i is the evolving context pool after each context change).

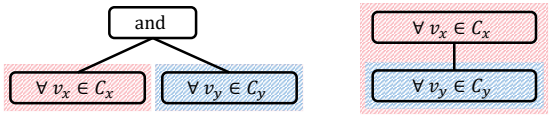
Incremental checking [10] examines each context change to analyze its impact on a constraint's previous checking result, while concurrent checking [11] would request multiple context changes for parallelism. In the following, we analyze the challenges when one combines the two techniques together.

B. 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). Suppose that initially robot r_1 is in warehouse x and r_2 in y. Then, 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. These robot movements induced a total of six context changes, with $<-, C_{\mathsf{y}}, \mathsf{r}_3>$ (chg') missed (five changes remaining), as illustrated in Fig. 1 (such missing-read events could be common in RFID-enabled sensing [13]–[16]).

When one conducts constraint checking on the context pool *upon each context change* (as the *individual checking* illustrates in Fig. 1) against the aforementioned $\mathsf{S_{loc}}$ 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.

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. 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.



(a) Parallel structure (b) Nested structure

Fig. 2. Two structures of consistency constraints

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.2a 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.2b, 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*?

C. 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, ...), P_i represents the evolving context pool after applying change chg_i to existing contexts in pool P_{i-1} (let P_0 be the initial pool). We use $\mathsf{chk}(P_i, s)$ and $\mathsf{Fuse\text{-}chk}(P_i, s)$ to denote the results of the ideal checking and our fusion checking 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 ($T = (chg_m, chg_{m+1}, ..., chg_n)$), 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 [5], incremental [10], or concurrent checking [11]). It is as follows:

$$\mathsf{Fuse\text{-}chk}(P_n, s) = \mathsf{chk}(P_n, s) \tag{2}$$

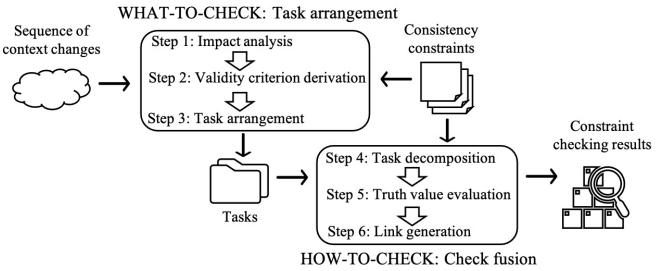


Fig. 3. Overview of our INFUSE approach

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

III. METHODOLOGY

A. 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 III-B), and the second part realizes the fusion of incremental and concurrent checking (Section III-C).

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.

B. 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 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* >, i.e., with a certain truth value.

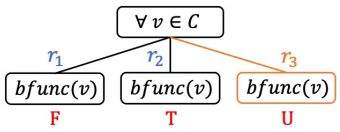


Fig. 4. Example of a universal formula.

TABLE I
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}}

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 II-A). Change <−, C_y, r_2> directly affects the constraint's existential quantifier part ($\exists v_y \in C_y$) and makes formula $\exists v_y \in C_y(\mathsf{Same}(v_x, v_y))$ 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 I 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.

Auxiliary functions:
- impact, where impact(chg, f) refers to chg's impact on f.
- base_impact, where base_impact(chg, ∃/∀) follows Table I.
- flip, where flip(m_{TT}) := m_{FF}; flip(m_{FF}) := m_{TT}; flip(m_{TF}) := m_{FT}; flip(m_{FT}) := m_{TF};
- flipSet, where flipSet(M) := {flip(m) | $m \in M$}.

Tracking rules:
- impact(chg, $\forall v \in C(f)$) =
  (1) base_impact(chg, ∀), when chg affects C,
  (2) impact(chg, f) ∪ {m_{FF}}, when chg affects f;
- impact(chg, $\exists v \in C(f)$) =
  (1) base_impact(chg, ∃), when chg affects C,
  (2) impact(chg, f) ∪ {m_{TT}}, when chg affects f;
- impact(chg, not (f)) = flipSet(impact(chg, f));
- impact(chg, (f_1) and (f_2)) =
  (1) impact(chg, f_1) ∪ {m_{FF}}, when chg affects f_1,
  (2) impact(chg, f_2) ∪ {m_{FF}}, when chg affects f_2;
- impact(chg, (f_1) or (f_2)) =
  (1) impact(chg, f_1) ∪ {m_{TT}}, when chg affects f_1,
  (2) impact(chg, f_2) ∪ {m_{TT}}, when chg affects f_2;
- impact(chg, (f_1) implies (f_2)) =
  (1) flipSet(impact(chg, f_1)) ∪{m_{TT}}, when chg affects f_1,
  (2) impact(chg, f_2) ∪ {m_{TT}}, when chg affects f_2.

Fig. 5. Tracking rules

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 impact(f) ∪{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 $\mathsf{S_{loc}}$ as follows:

$$\begin{aligned}
&\mathsf{impact}(\mathsf{chg}_1, \forall v_\mathsf{x} \in C_\mathsf{x}(\mathsf{not}(\exists v_\mathsf{y} \in C_\mathsf{y}(\mathsf{Same}(v_\mathsf{x}, v_\mathsf{y})))))\\
&= \mathsf{impact}(\mathsf{chg}_1, \mathsf{not}(\exists v_\mathsf{y} \in C_\mathsf{y}(\mathsf{Same}(v_\mathsf{x}, v_\mathsf{y})))) \cup \{\mathsf{m_{FF}}\}\\
&= \mathsf{flipSet}(\mathsf{impact}(\mathsf{chg}_1, \exists v_\mathsf{y} \in C_\mathsf{y}(\mathsf{Same}(v_\mathsf{x}, v_\mathsf{y})))) \cup \{\mathsf{m_{FF}}\}\\
&= \mathsf{flipSet}(\mathsf{base_impact}(\mathsf{chg}_1, \exists)) \cup \{\mathsf{m_{FF}}\}\\
&= \mathsf{flipSet}(\{\mathsf{m_{TT}}, \mathsf{m_{FT}}, \mathsf{m_{FF}}\}) \cup \{\mathsf{m_{FF}}\}\\
&= \{\mathsf{m_{FF}}, \mathsf{m_{TF}}, \mathsf{m_{TT}}\}
\end{aligned}$$

Similarly, the overall impacts of changes chg_2, chg_3, chg_4, and chg_5 in Fig. 1 can be obtained, 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 I), 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 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 $\mathsf{m_{TF}}$ impact (i.e., E-change), change chg_5 (<−, C_y, U>) has the $\mathsf{m_{FT}}$ impact (i.e., H-change), and the remaining change chg_2 has neither of them (i.e., I-change).

Then, consider two tasks: $T_1 = (\mathsf{chg}_1, \mathsf{chg}_2, \mathsf{chg}_3, \mathsf{chg}_4, \mathsf{chg}_5)$, and $T_2 = (\mathsf{chg}_1, \mathsf{chg}_2, \mathsf{chg}_3, \mathsf{chg}_4)$. 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 II-B.

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.

Algorithm 1: Task arrangement

```
   Input : set of consistency constraints S, new context
           change chg_new
   Output: set of consistency constraints S (updated)
1  for each s ∈ S do
2      p = impact (chg_new, s);
3      if p contains m_FT then
4          chg_new.type = H-change;
5      else if p contains m_TF then
6          chg_new.type = E-change;
7      else
8          chg_new.type = I-change;
9      if chg_new.type == H-change then
10         for each change chg in s.Task do
11             if chg.type == E-change then
12                 fusionchecking(s.Task, s);
13                 s.Task.clear();
14                 break;
15     s.Task ← append(chg_new);
16 return S;
```

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.*

Sketch of proof The theorem can be proved by reduction to absurdity, showing that each result in individual checking should be a subset of the final result of whole checking for any INFUSE-composed constraint checking task, because otherwise, the validity criterion would be broken. The detailed proof is at our website [17]. □

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

C. 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

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

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

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: $ASet = C' - C$, $DSet = C - C'$, and $USet = \{e | e \in C \cap C' \wedge \exists\ chg \in T(chg =< +/-, C, e >)\}$.

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). 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 [10]). The following gives an example link for our preceding inconsistency detected in the illustrative example (interesting readers can obtain more comprehensive explanations to links at our website [17]): $(\mathsf{violated}, \{(v_{\mathsf{x}} = \mathsf{r}_3), (v_{\mathsf{y}} = \mathsf{r}_3)\})$.

Step 5: Truth value evaluation. We use $\tau_{\mathsf{INFUSE}}[s]$ 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., $\tau_{\mathsf{INFUSE}}[s] = \tau_{\mathsf{partial}}[s]_\alpha$. 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. Due to the page limit, 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 website [17].

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

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

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

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

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 on 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,

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

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

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

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

context C, then the evaluation results of the remaining elements in C (i.e., $C - (ASet \cup USet)$) 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)).
(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., $C - (ASet \cup USet)$) 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 ("$\|$" means concurrent and ";" means sequential), since these evaluations are independent of each other.

We consider a universal or existential formula with its context affected by changes a concurrent point, which would incur the invocation of the $\mathsf{eval}_{\mathsf{entire}}$ or $\mathsf{eval}_{\mathsf{partial}}$ function and also feasible for initiating the concurrent checking in INFUSE.

For our preceding constraint $\mathsf{S}_{\mathsf{loc}}$ and a checking task $T = (\mathsf{chg}_1, \mathsf{chg}_2, \mathsf{chg}_2, \mathsf{chg}_4)$, these changes affect both the universal formula (i.e., $\forall v_x \in C_x$) and the inner existential formula (i.e., $\exists v_y \in C_y$) in $\mathsf{S}_{\mathsf{loc}}$. They are both candidates for initiating concurrent checking. INFUSE can choose both or the outermost one for the cost concern.

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

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

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

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 (recall our preceding example of link $(\mathsf{violated}, \{(v_\mathsf{x}, \mathsf{r}_3), (v_\mathsf{y}, \mathsf{r}_3)\})$). 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 website [17]).

(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 $\mathsf{gen}_{\mathsf{entire}}$ function in Fig. 11 and and $\mathcal{L}_{\mathsf{entire}}$ semantics in Fig. 10.
(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 $\mathsf{gen}_{\mathsf{entire}}$

function similarly.

(4) If the changes affect the subformula only, the link results of all elements should be updated by the $\mathsf{gen}_{\mathsf{partial}}$ 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 $\mathsf{gen}_{\mathsf{partial}}$ function and those of changed elements by the $\mathsf{gen}_{\mathsf{entire}}$ function.

Similarly, the $\mathsf{gen}_{\mathsf{entire}}$ and $\mathsf{gen}_{\mathsf{partial}}$ 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.*

Sketch of proof The complete proof is tedious. Basically, we prove that INFUSE works the same in terms of checking results (i.e., truth values and generated links) as full checking (ECC [5]), incremental checking (PCC [10]), and concurrent checking (Con-C [11]), for all seven formula types. The detailed proof is at our website [17]. □

As a summary, INFUSE conducts constraint checking with WHAT-Correctness for concurrency maximization and HOW-Correctness for fusion soundness. We next evaluate how this effort brings efficiency improvement.

IV. EVALUATION

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

A. Research Questions

W 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 (Practical Usage)**: *How effective is* INFUSE *in constraint checking under real-life settings?*

B. Experimental Design and Setup

Application. For fair comparisons, we used the taxi application, SmartCity, as our experimental subject, following existing work [9]–[12]. The application used massive taxi-driving data for smart route guidance.

Contexts. The application was accompanied with data concerning 2,716 vehicles monitored in a continuous period of 24 hours, i.e., 4.3 million raw driving data lines (containing vehicle id, GPS coordinates, driving speed and orientation, and service status). They correspond to 25.6 million *context changes* as modeled in the application, with varying workloads across different hours. To alleviate the experimental cost, we selected three distinct groups of data with *light*, *median*, and *heavy* workloads, representing the hours of 4am–5am (311,240 context changes), 9am–10am (843,686 changes), and 5pm–6pm (1,664,900 changes), respectively. The average intervals between consecutive changes are 11.6, 4.3, and 2.2 milliseconds (ms), respectively.

Constraints. We used all 48 consistency constraints associated with the application, also studied in existing work [9], [10]. They cover all formula types in the constraint language.

Process. In experiments, contexts are fed to the application with a middleware layer in between, which checks the contexts for consistency. We compare INFUSE with existing constraint checking techniques (ECC, PCC, and Con-C), using both their original versions (subscript "O") [10], [11] and and variants enhanced by GEAS (subscript "G") [9]. 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.* We use three constraint checking workloads, namely, light, median, and heavy, as aforementioned.
- *Running mode.* We study two running modes, namely, *offline* and *online*. With the former, next context changes are fed only when previous changes have been handled (comparing true efficiency differences). With the latter, context changes are fed strictly according to their timestamps, no matter whether previous changes have been handled or not (possibly causing false negatives or positives).

We design three dependent variables:

- *Checking time.* It measures the total time spent on constraint checking.
- *False negative rate (R_{FN}).* It measures the proportion of missed context inconsistencies against all inconsistencies that should be reported.
- *False positive rate (R_{FP}).* It measures the proportion of wrong context inconsistencies against all reported inconsistencies.

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

To answer RQ1, we compare six existing constraint checking techniques and INFUSE_0 on the heavy-workload contexts under the offline mode to observe their performance. To answer RQ2, we compare all eight constraints checking techniques on all three workload contexts under the offline mode, measuring the checking quality (by reported inconsistencies) and efficiency (by checking time). To answer RQ3, we compare all eight constraints checking techniques on all three workload contexts under the online mode (real-life

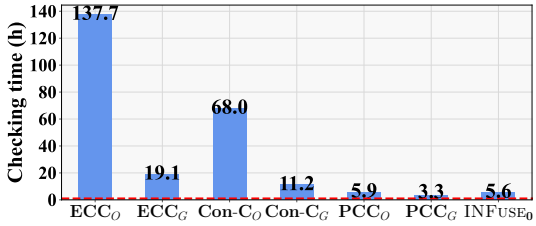


Fig. 12. Checking time comparison for the seven techniques on the heavy-workload contexts (the red dashed line represents the one-hour limit)

timestamps), measuring the checking quality (by false negative and positive rates) and efficiency (by checking time).

C. Experimental Results

We answer the three research questions in turn.

1) RQ1 (Motivation): We compared the checking time of the seven constraint checking techniques on the heavy-workload contexts in Fig. 12.

We observe that the checking time varied significantly for different constraint checking techniques, e.g., ECC up to 19.1–137.7 hours, Con-C for 11.2–68.0 hours, and PCC for 3.3–5.9 hours. We note that the time limit for handling this hour of contexts is *one hour only*, as illustrated by the red dashed line, and thus none of these techniques fulfilled the requirement, e.g., the worst case of ECC$_O$ took nearly six days! This strongly calls for more efficient constraint checking techniques. Besides, as INFUSE$_0$ shows, directly splitting incremental checking into parallel computing units did not bring significant improvement, behaving even worse than PCC$_G$.

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

2) RQ2 (Effectiveness): We compared the checking time of INFUSE and the seven techniques on all three workload contexts in Fig. 13. As the comparison was under the offline mode, all context changes were safely checked in turn, and thus all techniques obtained correct inconsistency detection results (this may not be true for the online mode later). Therefore, we focus on the checking time comparison here.

From the figure, we observe that: (1) although different workloads brought greatly varying checking time (from seconds to hours, almost 500x difference), INFUSE behaved significant and stable efficiency improvement for all workload contexts (always most efficient), e.g., 0.0x–18.6x improvement for the light workload, 2.4x–105.4x for median, and 3.1x–171.1x for heavy; (2) for all three workloads, INFUSE's checking time kept satisfactory (5.7 seconds, 7.7 minutes, and 0.8 hours, respectively), less than the one-hour limit; (3) with the growth of the checking workload, INFUSE exhibited increasing superiority over all other techniques, e.g., from an efficiency improvement up to 18.6x, to 105.4x, and to 171.1x, which is impressive; (4) when comparing INFUSE with the naïve implementation INFUSE$_0$, their difference was large and kept increasing, e.g., 5.7 s vs. 8.4 s (67.9%), 7.7 min vs. 46.7 min (16.5%), and 0.8 h vs. 5.6 h (14.3%). We owe all these achievements to INFUSE's concurrency maximization and fusion soundness.

Therefore, we answer RQ2 as follows: INFUSE *worked significantly efficient, achieving up to 18.6x, 105.4x, and 171.1x improvements for different workloads, as compared with existing constraint checking techniques.*

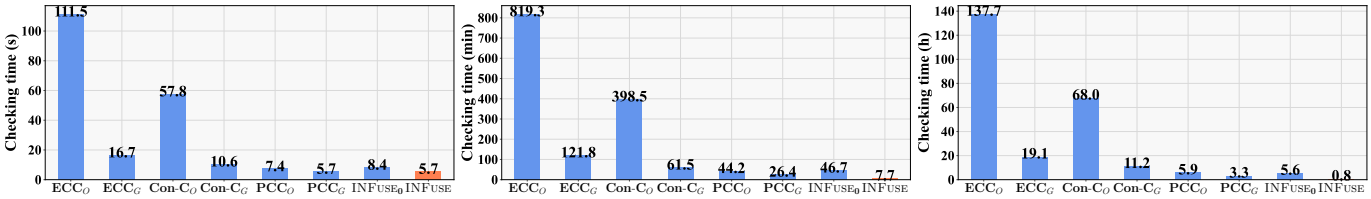
3) RQ3 (Practical Usage): We also compared INFUSE with the other seven techniques under an online mode, which simulated real-life context change scenarios. We focus on the checking quality (by false negative and positive rates R_{FN} and R_{FP}) and efficiency (by checking time). Table II lists the comparison results.

TABLE II
COMPARISONS AMONG ALL TECHNIQUES UNDER THE ONLINE MODE.

Workload	Checking techniques	Oracle incs (#)	Reported incs/* (#)	T_{cost}(s)	R_{FN}(%)	R_{FP}(%)
Light	ECC$_O$	3,254	3,254	128.6	0.0%	0.0%
	Con-C$_O$		3,254	54.3	0.0%	0.0%
	PCC$_O$		3,254	12.8	0.0%	0.0%
	ECC$_G$		3,254	26.9	0.0%	0.0%
	Con-C$_G$		3,254	16.9	0.0%	0.0%
	PCC$_G$		3,254	13.1	0.0%	0.0%
	INFUSE$_0$		3,254	13.1	0.0%	0.0%
	INFUSE		3,254	10.8	0.0%	0.0%
Median	ECC$_O$	21,436	8,647/694*	3,850.9	96.8%	92.0%
	Con-C$_O$		14,209/897*	3,593.9	95.8%	93.7%
	PCC$_O$		20,942/19,369*	1,513.7	9.6%	7.5%
	ECC$_G$		20,412/1,415*	3,588.4	93.4%	93.1%
	Con-C$_G$		20,779/19,293*	1,950.8	10.0%	7.2%
	PCC$_G$		21,377/19,414*	1,099.7	9.4%	9.2%
	INFUSE$_0$		20,922/19,371*	1,588.5	9.6%	7.4%
	INFUSE		21,436	456.6	0.0%	0.0%
Heavy	ECC$_O$	29,642	4,934/392*	4,032.1	98.7%	92.1%
	Con-C$_O$		6,611/463*	3,748.2	98.4%	93.0%
	PCC$_O$		22,574/1,028*	3,410.8	96.5%	95.5%
	ECC$_G$		14,617/801*	3,574.8	97.3%	94.5%
	Con-C$_G$		20,824/957*	3,375.5	96.8%	95.4%
	PCC$_G$		29,115/1,178*	3,594.4	96.0%	96.0%
	INFUSE$_0$		22,302/1,013*	3,463.2	96.6%	95.5%
	INFUSE		29,642	2,954.6	0.0%	0.0%

* represents the number of true positives among reported inconsistencies. If the slash "/" is omitted, all reported inconsistencies are true positives.

From the table, we observe that: (1) For the light workload, all checking techniques reported correct inconsistency results, but INFUSE took the least time, 10.8 seconds, 17.6–91.6% less than other techniques; (2) For the median workload, ECC$_O$, Con-C$_O$, and ECC$_G$ were subject to severe quality problems with over 90% false negative and positive rates, and PCC$_O$, Con-C$_G$, PCC$_G$, and INFUSE$_0$ suffered moderate quality problems with around 7%–10% false negative and positive rates, while INFUSE behaved perfectly with both zero false negative and positive rates, by taking still the least time; (3) For the heavy workload, all checking techniques took much more time, but still produced even worse results (92%–99% false negative and positive rates), except INFUSE, which achieved an amazing victory of still both zero false negative and positive rates. This suggests INFUSE's highly stable performance under very high workloads.



(a) Light workload (unit: second or s) (b) Median workload (unit: minute or min) (c) Heavy workload (unit: hour or h)

Fig. 13. Checking time comparison for all checking techniques on all workload contexts.

Therefore, we answer RQ3 as follows: INFUSE *worked still significantly efficient under real-life dynamic scenarios with zero false positive and negative, while other techniques could miss up to 98.7% and misreport up to 96.0% inconsistencies.*

D. Threats Analyses and Discussion

Although only one application was studied in the experiments, we tried to alleviate the concerned threats: (1) The application was also used in existing work [9]–[12], with the same set of consistency constraints and contexts to facilitate comparisons (fair); (2) We used all 48 consistency constraints, which cover all those used in existing work's experiments (comprehensive), and these constraints cover all formula types in the constraint language (complete); (3) Three groups of contexts were selected to represent different workloads to examine different constraint checking techniques (representative). Besides, to avoid possible bias, we (re)implemented all constraint checking techniques under the same I/O interface and data structures. We would later release our implementations to facilitate follow-up research.

V. Related Work

Our software engineering community has extensively studied the problem of consistency management for software artifacts, which could involve different development processes, e.g., software refactoring [18], method name suggestion [19], agile model-based development [20], or the whole software engineering process [21]. Some pieces of work focuses on managing the consistency of traditional software artifacts, like edit scripts [1], UML models [2]–[4], XML documents [5]–[7], and distributed source code [22], which are featured as being typically static or evolving slowly. Others tackle more dynamic artifacts in context-aware systems [23], attention-aware systems [24], and safety-critical systems [25]. Recently, the latter line of work receives increasing attention, and we are working along it with extensive application scenarios like Pollen Wise [26], Humanoid Companion Robot [27], and self-driving vehicle systems [28], [29]. Besides detecting inconsistencies in software artifacts, a relevant aspect of efforts is around resolving the inconsistencies by heuristics [30]–[32] and fixing strategies [33]–[37]. This also boosts the development of accompanying frameworks or supporting infrastructures like Cabot [38], Adam [39], and Lime [40].

Our work in this paper focuses on efficiently and effectively detecting inconsistencies in dynamic application contexts. On this particular aspect, various techniques work with varying efficiency gains and costs. For example, xlinkit [5], works in a full checking way, as the correctness baseline; PCC [10] checks incrementally by reusing previous results; Con-C [11] checks concurrently on units with similar workloads. All these are useful but gradually becoming less effective, with the continuous growth of environmental dynamics and context volume. Regarding this, GEAS [9] was proposed to cleverly schedule the checking of multiple context changes together to help accelerate a spectrum of existing techniques. Our work resembles this line, 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 show. INFUSE's idea opens a new direction to further improve constraint checking techniques. Our work exactly works along this line, trying to wisely fuse existing incremental and concurrent checking for even higher efficiency and better practical usages.

VI. Conclusion

In this work, we studied the efficient context inconsistency detection problem. We proposed a novel INFUSE approach, which on one hand automatically identifies valid context change groups for concurrency maximization, and on the other hand soundly fuses incremental and concurrent checking for reuse maximization. This effort works on both the constraint checking level and checking scheduling level, thus outperforming any existing constraint checking technique and checking scheduling strategy, as well as their direct combinations, realizing an 18.6x–171.1x efficiency improvement with quality guarantees. In future, we plan to more extensively validate INFUSE on comprehensive application scenarios, and explore possible finer-granularity balancing tunning inside the fusion checking for unexpected workload dynamics.

Acknowledgment

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 No. BK20202001. The authors would also like to thank the support from the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

REFERENCES

[1] T. Kehrer, U. Kelter, and G. Taentzer, "Consistency-preserving edit scripts in model versioning," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, E. Denney, T. Bultan, and A. Zeller, Eds. IEEE, 2013, pp. 191–201. [Online]. Available: https://doi.org/10.1109/ASE.2013.6693079

[2] R. S. Bashir, S. P. Lee, S. ur Rehman Khan, V. Chang, and S. Farid, "UML models consistency management: Guidelines for software quality manager," *Int. J. Inf. Manag.*, vol. 36, no. 6, pp. 883–899, 2016. [Online]. Available: https://doi.org/10.1016/j.ijinfomgt.2016.05.024

[3] N. Messaoudi, A. Chaoui, and M. Bettaz, "An approach to UML consistency checking based on compositional semantics," *Int. J. Embed. Real Time Commun. Syst.*, vol. 8, no. 2, pp. 1–23, 2017. [Online]. Available: https://doi.org/10.4018/IJERTCS.2017070101

[4] B. Wei and J. Sun, "Leveraging SPARQL queries for UML consistency checking," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 31, no. 4, pp. 635–654, 2021. [Online]. Available: https://doi.org/10.1142/S0218194021500170

[5] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein, "xlinkit: a consistency checking and smart link generation service," *ACM Trans. Internet Techn.*, vol. 2, no. 2, pp. 151–185, 2002. [Online]. Available: https://doi.org/10.1145/514183.514186

[6] S. P. Reiss, "Incremental maintenance of software artifacts," *IEEE Trans. Software Eng.*, vol. 32, no. 9, pp. 682–697, 2006. [Online]. Available: https://doi.org/10.1109/TSE.2006.91

[7] H. A. H. Handley, W. Khallouli, J. Huang, W. Edmonson, and N. Kibret, "Maintaining the consistency of sysml model exports to XML metadata interchange (XMI)," in *IEEE International Systems Conference, SysCon 2021, Vancouver, BC, Canada, April 15 - May 15, 2021*. IEEE, 2021, pp. 1–8. [Online]. Available: https://doi.org/10.1109/SysCon48628.2021.9447105

[8] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Proactive detection of collaboration conflicts," in *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*, T. Gyimóthy and A. Zeller, Eds. ACM, 2011, pp. 168–178. [Online]. Available: https://doi.org/10.1145/2025113.2025139

[9] H. Wang, C. Xu, B. Guo, X. Ma, and J. Lu, "Generic adaptive scheduling for efficient context inconsistency detection," *IEEE Trans. Software Eng.*, vol. 47, no. 3, pp. 464–497, 2021. [Online]. Available: https://doi.org/10.1109/TSE.2019.2898976

[10] C. Xu, S. C. Cheung, W. K. Chan, and C. Ye, "Partial constraint checking for context consistency in pervasive computing," *ACM Trans. Softw. Eng. Methodol.*, vol. 19, no. 3, pp. 9:1–9:61, 2010. [Online]. Available: https://doi.org/10.1145/1656250.1656253

[11] C. Xu, Y. Liu, S. C. Cheung, C. Cao, and J. Lv, "Towards context consistency by concurrent checking for internetware applications," *Sci. China Inf. Sci.*, vol. 56, no. 8, pp. 1–20, 2013. [Online]. Available: https://doi.org/10.1007/s11432-013-4907-5

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

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

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

[15] K. Patil, V. Bansal, V. Dhateria, and S. Narayankhedkar, "Probable causes of rfid tag read unreliability in supermarkets and proposed solutions," in *International Conference on Information Processing*, 12 2015, pp. 392–397.

[16] N. Fescioglu-Ünver, S. H. Choi, D. Sheen, and S. R. T. Kumara, "RFID in production and service systems: Technology, applications and issues," *Inf. Syst. Frontiers*, vol. 17, no. 6, pp. 1369–1380, 2015. [Online]. Available: https://doi.org/10.1007/s10796-014-9518-1

[17] "INFUSE website," https://sth4infuse.github.io/.

[18] H. A. Le, T. Dao, and N. Truong, "A formal approach to checking consistency in software refactoring," *Mob. Networks Appl.*, vol. 22, no. 2, pp. 356–366, 2017. [Online]. Available: https://doi.org/10.1007/s11036-017-0807-z

[19] Y. Li, S. Wang, and T. N. Nguyen, "A context-based automated approach for method name consistency checking and suggestion," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 574–586. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00060

[20] R. Jongeling, F. Ciccozzi, A. Cicchetti, and J. Carlson, "Lightweight consistency checking for agile model-based development in practice," *J. Object Technol.*, vol. 18, no. 2, pp. 11:1–20, 2019. [Online]. Available: https://doi.org/10.5381/jot.2019.18.2.a11

[21] C. Mayr-Dorn, R. Kretschmer, A. Egyed, R. Heradio, and D. Fernández-Amorós, "Inconsistency-tolerating guidance for software engineering processes," in *43rd IEEE/ACM International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER) 2021, Madrid, Spain, May 25-28, 2021*. IEEE, 2021, pp. 6–10. [Online]. Available: https://doi.org/10.1109/ICSE-NIER52604.2021.00010

[22] A. Demuth, M. Riedl-Ehrenleitner, and A. Egyed, "Efficient detection of inconsistencies in a multi-developer engineering environment," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, D. Lo, S. Apel, and S. Khurshid, Eds. ACM, 2016, pp. 590–601. [Online]. Available: https://doi.org/10.1145/2970276.2970304

[23] Y. Limón, E. Bárcenas, E. Benítez-Guerrero, and G. Molero, "On the consistency of context-aware systems," *J. Intell. Fuzzy Syst.*, vol. 34, no. 5, pp. 3373–3383, 2018. [Online]. Available: https://doi.org/10.3233/JIFS-169518

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

[25] C. Mayr-Dorn, M. Vierhauser, S. Bichler, F. Keplinger, J. Cleland-Huang, A. Egyed, and T. Mehofer, "Supporting quality assurance with automated process-centric quality constraints checking," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 1298–1310. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00118

[26] "Pollen wise - what's in your air, when and where," [EB/OL], https://play.google.com/store/apps/details?id=com.PollenSense.PollenWise Accessed May 13, 2022.

[27] P. Kuo, S. Lin, J. Hu, and C. Huang, "Multi-sensor context-aware based chatbot model: An application of humanoid companion robot," *Sensors*, vol. 21, no. 15, p. 5132, 2021. [Online]. Available: https://doi.org/10.3390/s21155132

[28] "Waymo," https://waymo.com.

[29] "The numbers dont lie: Self-driving cars are getting good." https://www.wired.com/2017/02/california-dmv-autonomous-car-disengagement/.

[30] Y. Bu, T. Gu, X. Tao, J. Li, S. Chen, and J. Lu, "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, 2006, pp. 193–200. [Online]. Available: https://doi.org/10.1109/QSIC.2006.38

[31] C. Xu, S. Cheung, W. K. Chan, and C. Ye, "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, 2008, pp. 713–721. [Online]. Available: https://doi.org/10.1109/ICDCS.2008.46

[32] J. Chomicki, J. Lobo, and S. A. Naqvi, "Conflict resolution using logic programming," *IEEE Trans. Knowl. Data Eng.*, vol. 15, no. 1, pp. 244–249, 2003. [Online]. Available: https://doi.org/10.1109/TKDE.2003.1161596

[33] C. Chen, C. Ye, and H. Jacobsen, "Hybrid context inconsistency resolution for context-aware services," in *Ninth Annual IEEE International Conference on Pervasive Computing and Communications,*

PerCom 2011, 21-25 March 2011, Seattle, WA, USA, Proceedings. IEEE, 2011, pp. 10–19. [Online]. Available: https://doi.org/10.1109/PERCOM.2011.5767574

[34] R. Kretschmer, D. E. Khelladi, A. Demuth, R. E. Lopez-Herrejon, and A. Egyed, "From abstract to concrete repairs of model inconsistencies: An automated approach," in *24th Asia-Pacific Software Engineering Conference, APSEC 2017, Nanjing, China, December 4-8, 2017*, J. Lv, H. J. Zhang, M. Hinchey, and X. Liu, Eds. IEEE Computer Society, 2017, pp. 456–465. [Online]. Available: https://doi.org/10.1109/APSEC.2017.52

[35] C. Xu, S. C. Cheung, W. K. Chan, and C. Ye, "On impact-oriented automatic resolution of pervasive context inconsistency," in *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*, I. Crnkovic and A. Bertolino, Eds. ACM, 2007, pp. 569–572. [Online]. Available: https://doi.org/10.1145/1287624.1287712

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

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

[38] C. Xu, S. Cheung, C. Lo, K. Leung, and J. Wei, "Cabot: On the ontology for the middleware support of context-aware pervasive applications," in *Network and Parallel Computing, IFIP International Conference, NPC 2004, Wuhan, China, October 18-20, 2004, Proceedings*, ser. Lecture Notes in Computer Science, H. Jin, G. R. Gao, Z. Xu, and H. Chen, Eds., vol. 3222. Springer, 2004, pp. 568–575. [Online]. Available: https://doi.org/10.1007/978-3-540-30141-7_85

[39] C. Xu, S. C. Cheung, X. Ma, C. Cao, and J. Lu, "Adam: Identifying defects in context-aware adaptation," *J. Syst. Softw.*, vol. 85, no. 12, pp. 2812–2828, 2012. [Online]. Available: https://doi.org/10.1016/j.jss.2012.04.078

[40] A. L. Murphy, G. P. Picco, and G. Roman, "LIME: A coordination model and middleware supporting mobility of hosts and agents," *ACM Trans. Softw. Eng. Methodol.*, vol. 15, no. 3, pp. 279–328, 2006. [Online]. Available: https://doi.org/10.1145/1151695.1151698