SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •
Special Focus

August 2013, Vol. 56 082105:1–082105:20
doi: 10.1007/s11432-013-4907-5

Towards context consistency by concurrent checking for Internetware applications

XU Chang[1,2]*, LIU YePang[3], CHEUNG S. C.[3], CAO Chun[1,2] & LV Jian[1,2]

[1] *State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing* 210023*, China;*
[2] *Department of Computer Science and Technology, Nanjing University, Nanjing* 210023*, China;*
[3] *Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong, China*

Received January 10, 2013; accepted May 14, 2013

Abstract Internetware applications are emerging and being widely used. They can adapt their behavior based on environmental contexts and deliver smart services. These contexts can be subject to various noises, which cause them to be inaccurate, incomplete, or even to conflict with each other. This is known as context inconsistency problem. Context inconsistency can trigger unexpected behavior to applications, and therefore should be prevented. One promising approach is to check contexts against consistency constraints so as to detect the occurrences of context inconsistency at runtime. Existing techniques have attempted different ways to improve the checking efficiency or effectiveness with different trade-offs in space overhead or communication cost. However, none of them has exploited multi-core computing capability to systematically improve the checking efficiency. In this paper, we propose a novel concurrent checking technique Con-C to efficiently detect inconsistencies in huge volumes of dynamic contexts. Con-C derives checking subtasks for each consistency constraint based on its structure and semantics. It achieves this in a fully automated way, and at the same time can guarantee its derived checking subtasks to be persistently balanced. We evaluated Con-C by controlled experiments through a large-scale real-world application. It reported promising results that Con-C improved the checking efficiency by extra 57.0%, in addition to what had been gained by incremental checking.

Keywords concurrent checking, consistency constraint, context inconsistency pervasive computing, Internetware

Citation Xu C, Liu Y P, Cheung S C, et al. Towards context consistency by concurrent checking for Internetware applications. Sci China Inf Sci, 2013, 56: 082105(20), doi: 10.1007/s11432-013-4907-5

1 Introduction

Internetware applications [1–4] aim to bridge the gap between closed, static computing and open, dynamic environments. They perceive environmental changes, and make necessary adaptation to their application logics in order to deliver smart services. We model these environmental changes by *environmental contexts*, or *contexts* as a short form for ease of presentation in this paper. During the adaptation process, the consistency of contexts about application environments is critical. If these contexts contain inaccurate, incomplete, or even conflicting information (called *context inconsistency* [5–7]), the concerned

*Corresponding author (email: changxu@nju.edu.cn)

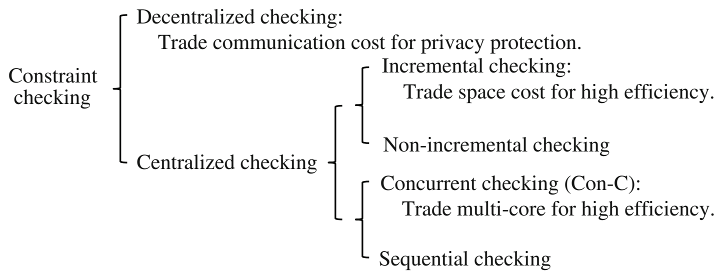
Constraint checking
Decentralized checking:
Trade communication cost for privacy protection.
Centralized checking
Incremental checking:
Trade space cost for high efficiency.
Non-incremental checking
Concurrent checking (Con-C):
Trade multi-core for high efficiency.
Sequential checking

Figure 1 Relationships among constraint checking techniques and their scopes.

applications can behave abnormally by taking unexpected adaptation, or even cause catastrophic consequences. Therefore, this gives rise to the research problem of maintaining context consistency for Internetware applications.

Within the scope of this paper, contexts can refer to any pieces of environmental information, like location, time, temperature, computer memory, and network bandwidth, as long as they affect the computing. Such contexts are typically subject to frequent changes and also vulnerable to various noises. For example, a typical RFID (Radio Frequency Identification technology [8]) application can easily make its object tracking functionality impaired by over 30% missing and cross read errors [9,10]. To detect inconsistencies from such uncontrollable noisy contexts, one may formulate physical laws or domain knowledge into consistency constraints [5–7] and check contexts against them. These constraints are supposed to specify necessary properties about contexts that must hold about application environments. When any constraint is violated, context inconsistency is detected. Then follow-up actions can be taken to resolve such inconsistency [5,11–13].

Inconsistency detection is the first step towards maintaining context consistency. In general, detecting inconsistencies in contexts resembles detecting inconsistencies in traditional software artifacts, like UML models [14–16], XML documents [17,18], and data structures [19,20]. However, the former further requires additional efforts in efficiency and effectiveness due to contexts' dynamic nature. Most existing techniques have worked in a centralized manner. That is, all software artifacts under checking are assumed to be at one host. They are checked against pre-specified consistency constraints in turn to expose each possible violation (inconsistency). Since contexts are dynamic, checking them for inconsistencies requires high efficiency. As such, incremental checking techniques [6,7] have been proposed to improve the checking efficiency by reusing previous results. They gain efficiency bonus with trade-off on extra space for storing previous results. Contexts also concern user privacy. This requires that distributed contexts on different hosts should not be transferred to one host for centralized checking. As such, decentralized checking [21] has also been studied for coordinating the checking among different hosts by exchanging checking subtasks and intermediate results. This reduces privacy threats with trade-off on extra communication overhead.

We thus observe that no technique can, and should, apply to all application scenarios. They suit selected target scenarios by fulfilling specialized goals with immaterial cost to those scenarios. However, a common feature of these techniques is that they have not exploited multi-core computing capability, which is nowadays becoming increasingly popular. We conjecture that concurrent checking by exploiting multi-core computing capability should be able to further improve the checking efficiency. This would also help reduce unnecessary power loss during multi-core running. Different from decentralized checking, concurrent checking still works at one host, and therefore does not incur any communication cost. Besides, concurrent checking can be complementary to incremental checking, as it improves the checking efficiency from a different dimension. We use Figure 1 to illustrate the relationships among these techniques and their scopes.

In this paper, we present a novel concurrent checking technique Con-C. Con-C carefully derives checking subtasks for each consistency constraint and makes all its computing units busy all the time. This is well known as the "balance" principle in distributed computing. However, we do not try from an engineering perspective by monitoring and scheduling computing tasks as traditional approaches do. Instead, we

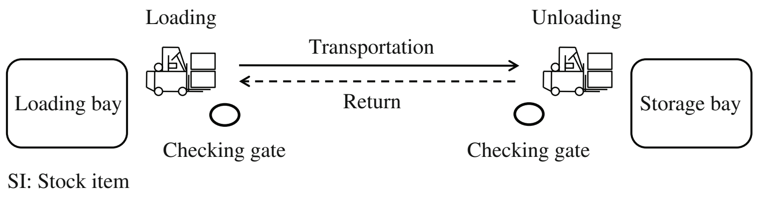

Figure 2 Stock tracking application scenario.

take an analysis perspective to derive checking subtasks based on each consistency constraint's structure and semantics. We make these derived checking subtasks always balanced as well as making the balance persistent. This nice property can guarantee high checking efficiency. It also enables Con-C applicable to different consistency constraints as well as making it transparent to users. We note that realizing this property is not straightforward. Consider n consistency constraints. One may intuitively choose to check each constraint individually and concurrently (e.g., by n threads). However, these n constraints can differ from each other and their checking workloads would also differ. Even for the same constraint, its checking workload would vary with time due to continual context changes. Therefore, this intuitive idea does not guarantee the checking to be balanced or persistently balanced. Our later evaluation also confirms that this idea can only bring limited efficiency gain. We shall explain how to realize persistently balanced checking in this paper. Highly-efficient concurrent context consistency checking would contribute to Internetware applications, by allowing them to effectively address dynamic contexts from increasingly larger Internet and Internet-of-things scenarios.

The remainder of this paper is organized as follows. Section 2 introduces background on context inconsistency detection through a stock tracking application. Section 3 presents Con-C data structures and explain how they support persistently balanced checking. Section 4 presents Con-C semantics and analyzes its correctness and efficiency. Section 5 evaluates Con-C through a large-scale real-world application. Section 6 discusses related work, and finally Section 7 concludes this paper.

2 Background

In this section, we present a stock tracking application. We use this application as example to explain how to detect context inconsistency by constraint checking.

2.1 Stock tracking application

The stock tracking application aims at automation for warehouse scenarios. Consider a warehouse, in which several forklifts repeatedly transport stock items from the loading bay of this warehouse to its storage bay, as illustrated in Figure 2. The application controls the workflow logic of each forklift. It decides what to do next based on its perceived contexts (e.g., when to pick up new stock items at the loading bay, when to start the transportation, and when to unload transported items at the storage bay). These contexts include locations of forklifts and their pallets (for containing stock items), and RFID reads about detected stock items when they pass by checking gates.

When the application resides at its "Unloading" state, its controlled forklift is supposed to unload all its transported stock items at the storage bay. Consider that RFID contexts can easily contain cross RFID reads or miss RFID reads. These collected reads should be checked at this state. For example, the following two consistency constraints can be specified for checking the validity and completeness of RFID reads (constraint C_1) and freeness of cross RFID reads (constraint C_2):

$$\begin{aligned}
C_1 : &(\forall\gamma_{\text{stor}} \text{ in STOR } (\text{valid}(\gamma_{\text{stor}}))) \text{ and} \\
&\qquad (\forall\gamma_{\text{load}} \text{ in LOAD } (\exists\gamma_{\text{stor}} \text{ in STOR } (\text{withinTrans}(\gamma_{\text{stor}},\ \gamma_{\text{load}})))),\\
C_2 : &\ \forall\gamma_{\text{stor}} \text{ in STOR } (\text{not } (\exists\gamma_{\text{dock}} \text{ in DOCK } (\text{withinTrans}(\gamma_{\text{stor}},\ \gamma_{\text{dock}})))).
\end{aligned}$$

The two constraints are specified in a first-order logic based language. Constraint C_1 requires that: 1) all RFID reads collected at the storage bay (i.e., set STOR) should be valid (by checking the "valid" function), and 2) any stock item detected at the loading bay (i.e., set LOAD) should be detected again later at the storage bay within the transportation time (by checking the "withinTrans" function). This constraint ensures the validity and completeness of all RFID reads about transported stock items. Constraint C_2 requires that all stock items should not be accidentally detected by other irrelevant RFID readers during the transportation. We assume that there is an irrelevant RFID reader near the forklift transportation route and its collected RFID reads are referred to as set DOCK. Those accidentally collected RFID reads that should not have been collected are called *cross reads*. This constraint ensures their nonexistence during the transportation. Detecting violation of such constraints (i.e., context inconsistency) is vital to the success of this application. Otherwise, it may obtain wrong inventory records or check out wrong stock items.

2.2 Constraint checking

Constraint checking needs to answer two basic questions: 1) Has a given consistency constraint been violated? 2) What has caused this violation? The processes of answering these two questions are *truth value evaluation* and *link generation*, respectively.

Given a consistency constraint, *truth value evaluation* decides whether this constraint is satisfied (when evaluated to true) or violated (when evaluated to false) with respect to the contexts under checking. *Link generation* further constructs a set of links for explaining why a constraint has been satisfied or violated [6,7,17]. These links connect certain contexts that contribute to this constraint's satisfaction or violation. These links essentially represent context inconsistencies. They contain useful information for guiding how context inconsistency should be resolved [11].

Discussing constraint checking needs to base on a certain constraint language. Within the scope of this paper, we assume the use of the following first-order logic based language. This language is expressive and can support consistency constraints for many existing applications including the stock tracking application. Still, we note that this language is only for our presentation, and our Con-C idea does not necessarily rely on this language. We give the language's syntax below:

$$\begin{aligned} f := \ & \forall\gamma \text{ in } S(f) \mid \exists\gamma \text{ in } S((f)) \mid \\ & (f) \text{ and } (f) \mid (f) \text{ or } (f) \mid (f) \text{ implies } (f) \mid \\ & \text{not } (f) \mid \mathit{bfunc}(\gamma, \ldots, \gamma). \end{aligned}$$

We briefly introduce this language. Any consistency constraint is specified by recursively using this syntax that contains seven formula types. Symbols "∀" and "∃" start a universal or existential formula, in which variable γ can take any value (context) from a set S (also called *context set*). Operators "and", "or", and "implies" start a formula with two parts of a conjunction, disjunction, or implication relation. Operator "not" starts a formula with a negation relation. Terminal *bfunc* can refer to any user-defined function that returns a Boolean value (true or false). The preceding stock tracking application has given two consistency constraint examples expressed in this language.

2.3 Truth value evaluation

Based on this constraint language, we explain truth value evaluation by elaborating its semantics [7] in Figure 3. In brief, $\mathcal{T}[f]_\alpha$ returns the truth value (true "⊤" or false "⊥") or formula f under variable assignment α. Here, *variable assignment* is a data structure that contains all mappings between variables and their assigned values (contexts). At the beginning of evaluating a consistency constraint, it starts with the constraint's top-level formula, accompanied with an empty variable assignment. With recursively analyzing this constraint's inner formulae, this variable assignment would be updated according to different semantics associated with different formula types. To facilitate accessing and manipulating variable assignments, we define two functions "get" and "bind". Function "get" retrieves the value of

$$\begin{array}{ll}
(1)\ \tau\,[\forall\gamma \text{ in } S\,(f)]_{\alpha} := & \top \wedge \tau\,[f]_{\text{bind}(\alpha,\,(\gamma,\,x1))} \wedge \ldots \wedge \tau\,[f]_{\text{bind}(\alpha,\,(\gamma,\,xn))} \mid x_i \in S. \\
(2)\ \tau\,[\exists\gamma \text{ in } S\,(f)]_{\alpha} := & \bot \vee \tau\,[f]_{\text{bind}(\alpha,\,(\gamma,\,x1))} \vee \ldots \vee \tau\,[f]_{\text{bind}(\alpha,\,(\gamma,\,xn))} \mid x_i \in S. \\
(3)\ \tau\,[(f_1) \text{ and } (f_2)]_{\alpha} := & \tau\,[f_1]_{\alpha} \wedge \tau\,[f_2]_{\alpha}. \\
(4)\ \tau\,[(f_1) \text{ or } (f_2)]_{\alpha} := & \tau\,[f_1]_{\alpha} \vee \tau\,[f_2]_{\alpha}. \\
(5)\ \tau\,[(f_1) \text{ implies } (f_2)]_{\alpha} := & \tau\,[f_1]_{\alpha} \rightarrow \tau\,[f_2]_{\alpha}. \\
(6)\ \tau\,[\text{not } (f)]_{\alpha} := & \leftarrow \tau\,[f]_{\alpha}. \\
(7)\ \tau\,[\mathit{bfunc}(\gamma_1, \ldots, \gamma_n)]_{\alpha} := & \mathit{bfunc}(\text{get}(\alpha, \gamma_1), \ldots, \text{get}(\alpha, \gamma_n)).
\end{array}$$

Figure 3 Truth value evaluation semantics.

a given variable γ from a variable assignment α: get(α, γ). Function "bind" adds a new variable-value mapping (γ, x) into a variable assignment α: bind(α, (γ, x)).

We explain some of these formulae as example. (1) Evaluating a universal formula "$\forall\gamma$ in $S(f)$" would consider its sub-formula f's truth value with variable γ's every possible value taken from S. If any value causes sub-formula f evaluated to false, the whole universal formula would also be evaluated to false. Note that we have also handled boundary cases when S is an empty set. (3) Evaluating formula "(f_1) and (f_2)" would consider the truth values of its both sub-formulae. If any sub-formula is evaluated to false, the whole "and" formula is also evaluated to false. (6) Evaluating formula "not (f)" would reverse its sub-formula f's truth value. (7) Evaluating formula "*bfunc*$(\gamma_1, \ldots, \gamma_n)$" would execute the user-defined function *bfunc* with concrete values of variables $\gamma_1, \ldots, \gamma_n$ retrieved from variable assignment α.

2.4 Link generation

Link generation would explain why consistency constraints have been satisfied or violated. We explain its idea by an example with our stocking tracking application. Suppose that during the transportation, a forklift collects three RFID reads at the loading bay, but only two at the storage bay. That is, one RFID read is missing at the storage bay. Formally, LOAD = $\{l_1, l_2, l_3\}$ and STOR = $\{s_1, s_2, s_3\}$. We consider the preceding constraint C_1. For ease of presentation, we assume all RFID reads at the storage bay to be valid, i.e., valid(s_1) = valid(s_3) = true. Then we only focus on the right part of constraint C_1 (we name it C_1'):

$$C_1' : \forall\gamma_{\text{load}} \text{ in LOAD } (\exists\gamma_{\text{stor}} \text{ in STOR } (\text{withinTrans}(\gamma_{\text{stor}}, \gamma_{\text{load}}))).$$

Constraint C_1' is violated because its truth value is false. In particular, for context l_2 in set LOAD, there does not exist any context γ_{stor} in STOR such that γ_{stor} can match l_2 by satisfying the "withinTrans" function. Then link generation would return the following result: $\{$(violated, (γ_{load}, l_2))$\}$.

This result contains one link: (violated, (γ_{load}, l_2)). It shows that constraint C_1' becomes violated when variable γ_{load} takes value l_2. This exactly explains the cause of context inconsistency, as well as suggesting possible resolution actions, e.g., restoring the missing RFID read back to set STOR.

We explain how such links are constructed by presenting the link generation semantics [7] in Figure 4. We discuss some formulae as example. (1) Consider a universal formula "$\forall\gamma$ in $S(f)$". If it is violated, there must exist at least one context x from S such that when variable γ is assigned with x, the sub-formula f is evaluated to false. Such context x is a reason why violation has occurred, and should be included into the links constructed for this universal formula. Here, a *concatenation* operator $\otimes$ is used to connect context x and other contexts used by sub-formula f that together explain this universal formula's violation. (3) Link generation for formula "(f_1) and (f_2)" would have to consider more cases. If both sub-formulae f_1 and f_2 are satisfied (Case i), the whole "and" formula is also satisfied. As the former is the necessary condition of the latter, links from both sub-formulae should be connected to together explain the "and" formula's satisfaction. For Case ii, as either of the two violated sub-formulae can lead to the "and" formula's violation, any link from both sub-formulae can explain the "and" formula's violation. This is achieved by a *merge* operator "$\cup$". The other two cases are similar. (6) Links for formula "not (f)" are constructed from links for its sub-formula f, except that the status is reversed. That is, the status changes from "satisfied" to "violated" and from "violated" to "satisfied" (by the "flipSet" function). This

$$
\begin{array}{lll}
(1)\ \mathcal{L}[\forall\gamma \text{ in } S\ (f)]_\alpha := & \multicolumn{2}{l}{\{l \mid l \in (\{(\text{violated}, (\gamma, x))\} \otimes \mathcal{L}[f]_{\text{bind}(\alpha, (\gamma, x))}) \wedge x{\in}S \wedge \tau\,[f]_{\text{bind}(\alpha, (\gamma, x))} = \bot\}.} \\
(2)\ \mathcal{L}[\exists\gamma \text{ in } S\ (f)]_\alpha := & \multicolumn{2}{l}{\{l \mid l \in (\{(\text{satisfied}, (\gamma, x))\} \otimes \mathcal{L}[f]_{\text{bind}(\alpha, (\gamma, x))}) \wedge x{\in}S \wedge \tau\,[f]_{\text{bind}(\alpha, (\gamma, x))} = \top\}.} \\
(3)\ \mathcal{L}[(f_1) \text{ and } (f_2)]_\alpha := & \text{i.} & \mathcal{L}[f_1]_\alpha \otimes \mathcal{L}[f_2]_\alpha, \text{if } \tau\,[f_1]_\alpha = \tau\,[f_2]_\alpha = \top; \\
 & \text{ii.} & \mathcal{L}[f_1]_\alpha \cup \mathcal{L}[f_2]_\alpha, \text{if } \tau\,[f_1]_\alpha = \tau\,[f_2]_\alpha = \bot; \\
 & \text{iii.} & \mathcal{L}[f_2]_\alpha, \text{if } \tau\,[f_1]_\alpha = \top \text{ and } \tau\,[f_2]_\alpha = \bot; \\
 & \text{iv.} & \mathcal{L}[f_1]_\alpha, \text{if } \tau\,[f_1]_\alpha = \bot \text{ and } \tau\,[f_2]_\alpha = \top. \\
(4)\ \mathcal{L}[(f_1) \text{ or } (f_2)]_\alpha := & \text{i.} & \mathcal{L}[f_1]_\alpha \cup \mathcal{L}[f_2]_\alpha, \text{if } \tau\,[f_1]_\alpha = \tau\,[f_2]_\alpha = \top; \\
 & \text{ii.} & \mathcal{L}[f_1]_\alpha \otimes \mathcal{L}[f_2]_\alpha, \text{if } \tau\,[f_1]_\alpha = \tau\,[f_2]_\alpha = \bot; \\
 & \text{iii.} & \mathcal{L}[f_1]_\alpha, \text{if } \tau\,[f_1]_\alpha = \top \text{ and } \tau\,[f_2]_\alpha = \bot; \\
 & \text{iv.} & \mathcal{L}[f_2]_\alpha, \text{if } \tau\,[f_1]_\alpha = \bot \text{ and } \tau\,[f_2]_\alpha = \top. \\
(5)\ \mathcal{L}[(f_1) \text{ implies } (f_2)]_\alpha := & \text{i.} & \text{flipSet}(\mathcal{L}[f_1]_\alpha) \otimes \mathcal{L}[f_2]_\alpha, \text{if } \tau\,[f_1]_\alpha = \top \text{ and } \tau\,[f_2]_\alpha = \bot; \\
 & \text{ii.} & \text{flipSet}(\mathcal{L}[f_1]_\alpha) \cup \mathcal{L}[f_2]_\alpha, \text{if } \tau\,[f_1]_\alpha = \bot \text{ and } \tau\,[f_2]_\alpha = \top; \\
 & \text{iii.} & \mathcal{L}[f_2]_\alpha, \text{if } \tau\,[f_1]_\alpha = \tau\,[f_2]_\alpha = \top; \\
 & \text{iv.} & \text{flipSet}(\mathcal{L}[f_1]_\alpha), \text{ if } \tau\,[f_1]_\alpha = \tau\,[f_2]_\alpha = \bot. \\
(6)\ \mathcal{L}[\text{not } (f)]_\alpha := & \multicolumn{2}{l}{\text{flipSet}(\mathcal{L}[f]_\alpha).} \\
(7)\ \mathcal{L}[\mathit{bfunc}(\gamma_1, \ldots, \gamma_n)]_\alpha := & \multicolumn{2}{l}{\varnothing.}
\end{array}
$$

Figure 4 Link generation semantics.

is because any link explaining why sub-formula f is satisfied/violated also explains why formula "not (f)" is violated/satisfied.

We note that the formal definitions of concatenation and merge operators are not the focus of this paper, and are therefore omitted. Interested readers can refer to our previous work [7] for a complete treatment and more examples.

3 Con-C data structures

The truth value evaluation and link generation semantics introduced in Section 2 do not take any short-circuit optimization. This is because context inconsistency detection aims to analyze all reasons that cause a consistency constraint violated, and then come up with actions to resolve all these problems. When short-circuit optimization is used, truth value evaluation and link generation would stop sooner by skipping analyzing some parts of a constraint. This would lead to incomplete inconsistency analysis results, which are undesirable. Therefore, inconsistency analysis does not take any short-circuit optimization [7,17]. Still, we note that these semantics work in a sequential manner, and do not exploit multi-core computing capability. As such, we in the following present our Con-C idea to make them work in a concurrent manner. First, we introduce our data structures supporting Con-C.

3.1 Syntax tree and runtime tree

Given a consistency constraint, its *syntax tree* shows how this constraint is syntactically constructed in a hierarchical way. Figure 5 illustrates syntax trees for our preceding two constraints C_1 and C_2. We can observe that a syntax tree is always static, and its structure is fully decided by its corresponding consistency constraint. It does not rely on the contexts under checking.

Given a consistency constraint, its *runtime tree* shows how this constraint is checked with respect to the contexts under checking. Suppose that context sets LOAD, STOR, and DOCK in constraints C_1 and C_2 take the following values:

$$\text{LOAD} := \{l_1, l_2\}, \quad \text{STOR} := \{s_1, s_2\}, \quad \text{DOCK} := \{d_1\}.$$

With respect to these values (contexts), Figure 6 illustrates constraints C_1's and C_2's associated runtime trees. We explain how they are constructed. A universal or existential formula "$\forall\gamma$ in $S(f)$"/"$\exists\gamma$ in $S(f)$" would take multiple branches, whose number is equal to the size of this formula's associated context set

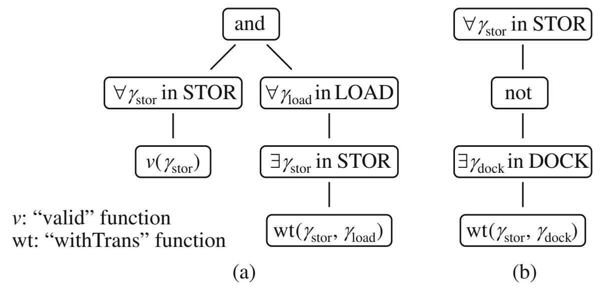
and
$\forall \gamma_{\text{stor}}$ in STOR
$\forall \gamma_{\text{load}}$ in LOAD
$v(\gamma_{\text{stor}})$
$\exists \gamma_{\text{stor}}$ in STOR
v: "valid" function
wt: "withTrans" function
$\text{wt}(\gamma_{\text{stor}}, \gamma_{\text{load}})$
(a)
$\forall \gamma_{\text{stor}}$ in STOR
not
$\exists \gamma_{\text{dock}}$ in DOCK
$\text{wt}(\gamma_{\text{stor}}, \gamma_{\text{dock}})$
(b)

Figure 5 Syntax tree examples. (a) Constraint C_1; (b) constraint C_2.

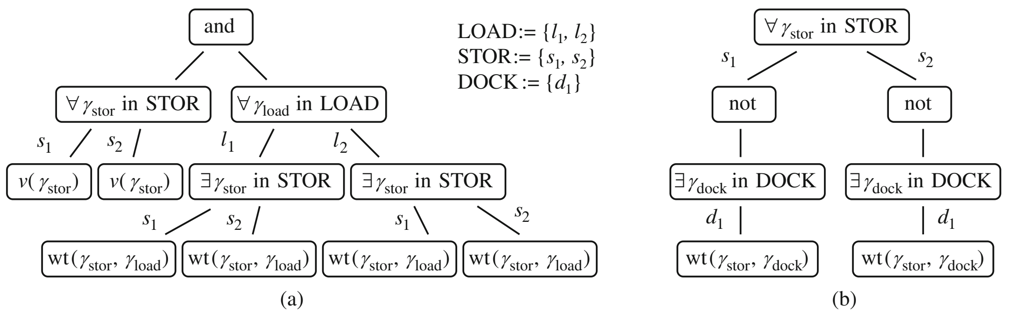
and
LOAD:= $\{l_1, l_2\}$
STOR:= $\{s_1, s_2\}$
DOCK := $\{d_1\}$
$\forall \gamma_{\text{stor}}$ in STOR
$\forall \gamma_{\text{load}}$ in LOAD
s_1
s_2
l_1
l_2
$v(\gamma_{\text{stor}})$
$v(\gamma_{\text{stor}})$
$\exists \gamma_{\text{stor}}$ in STOR
$\exists \gamma_{\text{stor}}$ in STOR
$\text{wt}(\gamma_{\text{stor}}, \gamma_{\text{load}})$
(a)
$\forall \gamma_{\text{stor}}$ in STOR
not
$\exists \gamma_{\text{dock}}$ in DOCK
d_1
$\text{wt}(\gamma_{\text{stor}}, \gamma_{\text{dock}})$
(b)

Figure 6 Runtime tree examples. (a) Constraint C_1; (b) constraint C_2.

S. Each branch is labeled with a particular context from S and assigned to this formula's associated variable γ. These branches represent this formula's sub-formula f with different variable assignments. An "and", "or", and "implies" formula would take two branches, each of which corresponds to one of its sub-formulae. A "not" formula takes one branch, which corresponds to its only sub-formula. Terminal *bfunc* is always a leaf node because it does not have any sub-formula.

We can observe that a runtime tree is dynamic, and its structure is decided by its corresponding syntax tree as well as the contexts under checking. When the contexts are subject to change, a runtime tree would change accordingly. However, this change only reflects at a runtime tree's width, but not at its height.

Runtime tree naturally supports constraint checking. A consistency constraint's truth value evaluation and link generation can be done by a post-order traversal of this constraint's associated runtime tree. This is because according to the truth value evaluation and link generation semantics, each formula's truth value and links depend only on those of its sub-formulae. This exactly corresponds to a post-order traversal of the whole constraint's associated runtime tree (visiting all child nodes before visiting itself).

3.2 Splitting node

Post-order traversal can achieve constraint checking, but it works in a sequential manner. We aim at realizing concurrent checking in a persistently balanced way. This requires us to find some special nodes on a runtime tree, at which we split checking tasks always in a balanced way. These nodes are called *persistently balanced splitting nodes*.

Our basic idea is: If the traversal of a runtime tree is conducted concurrently by multiple computing units, and the travelled parts by each computing unit contain the same number of, and same types of, nodes, then these computing units would carry balanced checking workloads.

We start our discussion with splitting node. Given a runtime tree, every node containing more than one branch is a *splitting node*. From a splitting node, multiple computing units can work together because every of its branches can be constructed and travelled by an individual computing unit. Since these branches have nothing overlapping (no dependency at all), all computing units can work concurrently without any interference to each other.

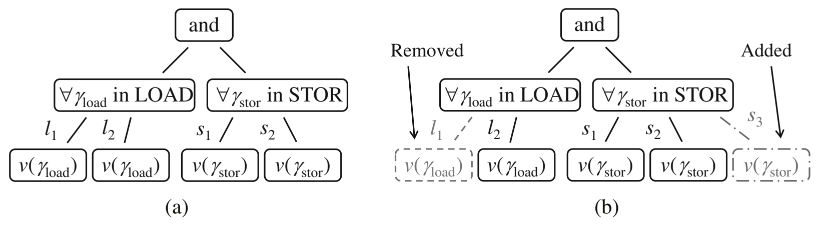
and
$\forall\gamma_{\text{load}}$ in LOAD
$\forall\gamma_{\text{stor}}$ in STOR
l_1
l_2
s_1
s_2
s_3
$v(\gamma_{\text{load}})$
$v(\gamma_{\text{stor}})$
Removed
Added
(a)
(b)

Figure 7 Runtime tree changes. (a) At time t_1; (b) at time t_2.

Splitting node is common in a runtime tree. For example, in Figure 6(a), the root node ("and" node), both universal nodes ("$\forall\gamma_{\text{stor}}$ in STOR" and "$\forall\gamma_{\text{load}}$ in LOAD" nodes), and both existential nodes ("$\exists\gamma_{\text{stor}}$ in STOR" node) are all splitting nodes for constraint C_1's runtime tree.

By splitting node, concurrent checking can work at a *formula* level (rather than at a *constraint* level). One part of constraint checking (below the selected splitting node) is divided into several pieces, and each piece is assigned to a computing unit for concurrent checking. Remaining parts of constraint checking (above the selected splitting node and at other branches) still need to be completed sequentially. This is common to general parallel computing.

3.3 Balanced splitting node

Splitting nodes may not be necessarily balanced. The checking workload (including branch construction and traversal) for one of its branches may differ from that for another branch. For example, the root node ("and" node) in Figure 6(a) is not a *balanced splitting node*. Its left branch contains much less nodes (three) than its right branch (seven). Besides, the two branches carry different types of nodes (two types vs. three different types). This leads to clearly different checking workloads when the two branches are constructed and travelled by two computing units.

On the other hand, we do have balanced splitting nodes in this runtime tree. For example, the right universal node ("$\forall\gamma_{\text{load}}$ in LOAD" node) in Figure 6(a) is a balanced splitting node. It has two branches, which contain exactly the same numbers of, and same types of, nodes (three nodes and two types). Then two computing units can undertake the same checking workload when constructing and travelling the two branches. Such balanced splitting nodes are desirable in concurrent checking.

3.4 Persistently balanced splitting node

Balanced splitting nodes may not be necessarily persistent. As mentioned earlier, the structure of a runtime tree depends also on the contexts under checking. Since the contexts are subject to change, the tree's structure would also change accordingly at runtime. As such, for a balanced splitting node, the checking workload for its every branch may not always be equal to each other all the time.

For example, consider a new consistency constraint C_3, which is slightly modified from our preceding constraint C_1. Constraint C_3 checks the validity of all RFID reads collected either at the loading bay or at the storage bay:

$$C_3 : (\forall\gamma_{\text{load}} \text{ in LOAD } (\text{valid}(\gamma_{\text{load}}))) \text{ and } (\forall\gamma_{\text{stor}} \text{ in STOR } (\text{valid}(\gamma_{\text{stor}}))).$$

Suppose that at time t_1, two context sets LOAD and STOR contain the following values:

$$\text{LOAD} := \{l_1, l_2\}, \quad \text{STOR} := \{s_1, s_2\}.$$

Based on these values, Figure 7(a) illustrates constraint C_3's runtime tree at time t_1. We can observe that the root node ("and" node) is a balanced splitting node, because its two branches contain the same number of, and same types of, nodes (three nodes and two types). However, this balance is not persistent. Suppose that at next time t_2, two context sets LOAD and STOR change to the following new values:

$$\text{LOAD} := \{l_2\}, \quad \text{STOR} := \{s_1, s_2, s_3\}.$$

It means that an existing context l_1 becomes obsolete and is removed from set LOAD, and a new context s_3 is collected and added into set STOR. Accordingly, this constraint's runtime tree changes to another form at time t_2, as illustrated in Figure 7(b). Now the root node ("and" node) is no longer a balanced splitting node, because its two branches contain different numbers of nodes (two nodes vs. four nodes). Therefore, we say that the root node ("and" node) is not a *persistently balanced splitting node* (or PB splitting node).

For a non-PB splitting node, its checking workload for different branches is not under control. It depends on the contexts under checking (sometimes balanced, but sometimes not). In concurrent checking, PB splitting nodes are desirable.

3.5 Properties

We discuss and prove some properties about PB splitting nodes.

Theorem 1. Given a consistency constraint and its associated runtime tree, nodes with "and", "or", and "implies" operators are not candidates for PB splitting nodes.

Proof. The three operators are similar, and we discuss "and" operator as example.

A node with "and" operator has two branches, which correspond to the "and" formula's two sub-formulae. There are two cases. 1) If the two sub-formulae have different structures in their syntax tree, the two branches would incur different workloads in constraint checking. Then the "and" node is not a candidate for PB splitting node. 2) Otherwise, the two sub-formulae have the same structure (same number of nodes and same node types). There are also two cases. (2a) If the two sub-formulae define and use different variables, the "and" node's balance (if any) is not persistent. This is because their corresponding two branches in the runtime tree would be subject to change when contexts change, as illustrated in Figure 7. Then the "and" node is not a candidate for PB splitting node. (2b) Otherwise, they define and use the same variables. Then the two sub-formulae are actually the same, and can be combined into one. In theory this "and" node is gone, and in practice this case does not occur. This completes the proof.

This theorem tells that one should not select nodes with "and"/"or"/"implies" operator as PB splitting nodes. They either are not balanced, or even if balanced, the balance is not persistent. Note that we are discussing general cases. Some consistency constraints may have their "and"/"or"/"implies" nodes persistently balanced in very special cases, and this is not our focus.

Theorem 2. Given a consistency constraint and its associated runtime tree, nodes with universal and existential operators are candidates for PB splitting nodes.

Proof. Universal and existential operators are similar, and we discuss universal operator as example.

Consider a node associated with universal operator. It corresponds to a universal formula in the syntax tree. Let the universal formula be f. The universal node contains multiple branches, and the number of these branches is decided by the number of contexts referred to by f's defined variable. According to the process of constructing a runtime tree, all these branches share the same structure but with different contexts assigned to f's defined variable. Similarly, if f's sub-formula further defines new variables (i.e., nested universal or existential formulae), they would keep sharing the same structure. As such, the universal node has the same structure for its every branch. Since this does not depend on the contexts under checking, this same-structure property holds all the time. Then the universal node is a candidate for PB splitting node. This completes the proof.

This theorem tells that one can select nodes with universal or existential operator as PB splitting nodes. Their balance is persistent.

Theorem 3. Given a consistency constraint and its associated runtime tree, there must exist at least one PB splitting node.

$$\begin{array}{ll}
(1)\ \tau[\forall\gamma \text{ in } S\ (f)]_{\alpha} := \{ & (t_1, \ldots, t_n) := \text{concurrent}(\tau[f]_{\text{bind}(\alpha,\,(\gamma,\,xi))} \mid x_i \in S); \\
 & \text{return } (\top \wedge t_1 \wedge \ldots \wedge t_n);\quad \}. \\
(2)\ \tau[\exists\gamma \text{ in } S\ (f)]_{\alpha} := \{ & (t_1, \ldots, t_n) := \text{concurrent}(\tau[f]_{\text{bind}(\alpha,\,(\gamma,\,xi))} \mid x_i \in S); \\
 & \text{return } (\bot \vee t_1 \vee \ldots \vee t_n);\quad \}. \\
(3)\ \tau[(f_1) \text{ and } (f_2)]_{\alpha} := & \tau[f_1]_{\alpha} \wedge \tau[f_2]_{\alpha}. \\
(4)\ \tau[(f_1) \text{ or } (f_2)]_{\alpha} := & \tau[f_1]_{\alpha} \vee \tau[f_2]_{\alpha}. \\
(5)\ \tau[(f_1) \text{ implies } (f_2)]_{\alpha} := & \tau[f_1]_{\alpha} \rightarrow \tau[f_2]_{\alpha}. \\
(6)\ \tau[\text{not } (f)]_{\alpha} := & \neg\,\tau[f]_{\alpha}. \\
(7)\ \tau[bfunc(\gamma_1, \ldots, \gamma_n)]_{\alpha} := & bfunc(\text{get}(\alpha, \gamma_1), \ldots, \text{get}(\alpha, \gamma_n)).
\end{array}$$

Figure 8 Con-C truth value evaluation semantics.

Proof. Each consistency constraint uses at least one variable (otherwise, it is a tautology or contradictory but not a constraint). Since such variable can only be defined in a universal or existential formula, there must exist at least one universal or existential node in this constraint's runtime tree. According to Theorem 2, this node is a PB splitting node. This completes the proof.

This theorem tells that given any consistency constraint, its runtime tree must own one PB splitting node. It is the place from which we start concurrent checking. We present detailed checking semantics below.

4 Concurrent checking

In this section, we present our new truth value evaluation and link generation semantics for concurrent constraint checking. For ease of presentation, we call them Con-C truth value evaluation and link generation, respectively.

4.1 Con-C semantics

As we discussed, any universal or existential node can be a PB splitting node. From this node, multiple computing units can start concurrent construction and traversal of its every branch in a persistently balanced way. Based on this observation, we give Con-C truth value evaluation semantics in Figure 8.

We first consider universal/existential formula. Keyword "concurrent" in Figure 8 assigns $|S|$ subtasks "$\tau[f]_{bind(\alpha,\,(\gamma,\,xi))}$" ($x_i \in S$) to n computing units for concurrent checking. This corresponds to a construction and traversal of this formula's n corresponding branches in the runtime tree concurrently. Then n calculated truth values from different computing units $(t_1, \ldots, t_n)$ are combined into one as the final result (by keyword "return"). Since other formulae cannot be selected as PB splitting nodes, they would follow their original truth value evaluation semantics.

We then give Con-C link generation semantics in Figure 9. They can be explained similarly, and we omit their elaboration.

4.2 Con-C correctness

Con-C checking semantics differ from their original checking semantics (named Seq-C). We need the following theorem to guarantee its correctness.

Theorem 4. Given a consistency constraint and a set of contexts checked against this constraint, Con-C always returns the same checking result as Seq-C does.

Proof. Con-C correctness is guaranteed by the equivalence between Con-C checking results and those of Seq-C. Among seven formula types, five of them have the same checking semantics for Con-C and Seq-C. Only universal and existential formulae differ.

Let us consider universal formula. For its truth value evaluation, Con-C splits the calculation into n parts ($n := |S|$), assigns them to n computing units, and then combines n partial results into the

(1) $\mathcal{L}[\forall\gamma \text{ in } S\ (f)]_\alpha := \{$ $(l_1, \ldots, l_m) := \text{concurrent}(\mathcal{L}[f]_{\text{bind}(\alpha,\ (\gamma,\ xi))} \mid x_i \in S \wedge \tau\ [f]_{\text{bind}(\alpha,\ (\gamma,\ xi))} = \bot);$
return $(\varnothing \cup (\{(\text{violated}, (\gamma, x))\} \otimes l_1) \cup \ldots \cup (\{(\text{violated}, (\gamma, x))\} \otimes l_m));$ $\}.$

(2) $\mathcal{L}[\exists\gamma \text{ in } S\ (f)]_\alpha := \{$ $(l_1, \ldots, l_m) := \text{concurrent}(\mathcal{L}[f]_{\text{bind}(\alpha,\ (\gamma,\ xi))} \mid x_i \in S \wedge \tau\ [f]_{\text{bind}(\alpha,\ (\gamma,\ xi))} = \top);$
return $(\varnothing \cup (\{(\text{satisfied}, (\gamma, x))\} \otimes l_1) \cup \ldots \cup (\{(\text{satisfied}, (\gamma, x))\} \otimes l_m));$ $\}.$

(3) $\mathcal{L}[(f_1) \text{ and } (f_2)]_\alpha :=$
i. $\mathcal{L}[f_1]_\alpha \otimes \mathcal{L}[f_2]_\alpha$, if $\tau\ [f_1]_\alpha = \tau\ [f_2]_\alpha = \top$;
ii. $\mathcal{L}[f_1]_\alpha \cup \mathcal{L}[f_2]_\alpha$, if $\tau\ [f_1]_\alpha = \tau\ [f_2]_\alpha = \bot$;
iii. $\mathcal{L}[f_2]_\alpha$, if $\tau\ [f_1]_\alpha = \top$ and $\tau\ [f_2]_\alpha = \bot$;
iv. $\mathcal{L}[f_1]_\alpha$, if $\tau\ [f_1]_\alpha = \bot$ and $\tau\ [f_2]_\alpha = \top$.

(4) $\mathcal{L}[(f_1) \text{ or } (f_2)]_\alpha :=$
i. $\mathcal{L}[f_1]_\alpha \cup \mathcal{L}[f_2]_\alpha$, if $\tau\ [f_1]_\alpha = \tau\ [f_2]_\alpha = \top$;
ii. $\mathcal{L}[f_1]_\alpha \otimes \mathcal{L}[f_2]_\alpha$, if $\tau\ [f_1]_\alpha = \tau\ [f_2]_\alpha = \bot$;
iii. $\mathcal{L}[f_1]_\alpha$, if $\tau\ [f_1]_\alpha = \top$ and $\tau\ [f_2]_\alpha = \bot$;
iv. $\mathcal{L}[f_2]_\alpha$, if $\tau\ [f_1]_\alpha = \bot$ and $\tau\ [f_2]_\alpha = \top$.

(5) $\mathcal{L}[(f_1) \text{ implies } (f_2)]_\alpha :=$
i. $\text{flipSet}(\mathcal{L}[f_1]_\alpha) \otimes \mathcal{L}[f_2]_\alpha$, if $\tau\ [f_1]_\alpha = \top$ and $\tau\ [f_2]_\alpha = \bot$;
ii. $\text{flipSet}(\mathcal{L}[f_1]_\alpha) \cup \mathcal{L}[f_2]_\alpha$, if $\tau\ [f_1]_\alpha = \bot$ and $\tau\ [f_2]_\alpha = \top$;
iii. $\mathcal{L}[f_2]_\alpha$, if $\tau\ [f_1]_\alpha = \tau\ [f_2]_\alpha = \top$;
iv. $\text{flipSet}(\mathcal{L}[f_1]_\alpha)$, if $\tau\ [f_1]_\alpha = \tau\ [f_2]_\alpha = \bot$.

(6) $\mathcal{L}[\text{not } (f)]_\alpha :=$ $\text{flipSet}(\mathcal{L}[f]_\alpha)$.

(7) $\mathcal{L}[\mathit{bfunc}(\gamma_1, \ldots, \gamma_n)]_\alpha :=$ $\varnothing$.

Figure 9 Con-C link generation semantics.

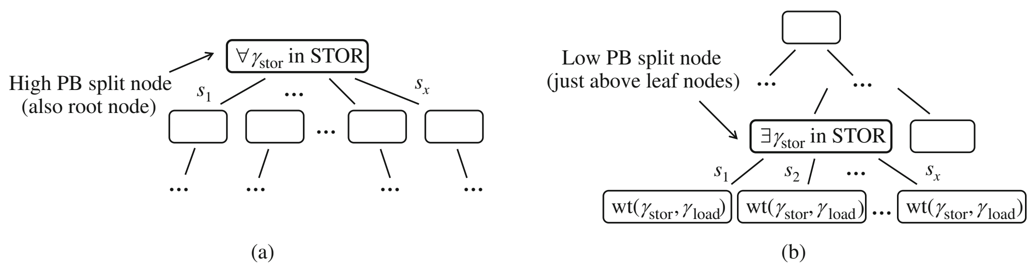
High PB split node
(also root node)
$\forall\gamma_{\text{stor}}$ in STOR
s_1
s_x
(a)
Low PB split node
(just above leaf nodes)
$\exists\gamma_{\text{stor}}$ in STOR
s_1
s_2
s_x
$\text{wt}(\gamma_{\text{stor}}, \gamma_{\text{load}})$
$\text{wt}(\gamma_{\text{stor}}, \gamma_{\text{load}})$
$\text{wt}(\gamma_{\text{stor}}, \gamma_{\text{load}})$
(b)

Figure 10 Speedup illustration.

final result. For its link generation, Con-C splits the calculation into m parts ($m := |\{x_i | x_i \in S \wedge \tau[f]_{bind(\alpha,\,(\gamma,\,xi))} = \bot\}|$), assigns them to m computing units, and then combines m partial results into the final result. As the calculation is essentially the same as that of Seq-C, Con-C returns the same result as Seq-C does for universal formula. Existential formula can be proved similarly.

Since other formula types use the same checking semantics in Con-C and Seq-C. Con-C for any consistency constraint would return the same checking result as Seq-C does. This completes the proof.

4.3 Con-C speedup

Con-C speeds up the construction and traversal of runtime trees by conducting them concurrently. Consider that a runtime tree may own more than one PB splitting node. Selecting any one from them suffices for Con-C. However, the selection choice may affect how much efficiency can be improved in constraint checking.

When the selected PB splitting node is as high as possible in a runtime tree (e.g., up to the root node that is also a universal or existential node), the checking workload for this tree would be evenly split into x parts. Here, x is the number of contexts that can be assigned to this universal or existential node's associated variable. In this case, the speedup rate can be close to x (i.e., almost $x - 1$ times faster), as illustrated in Figure 10(a).

When the selected PB splitting node is as low as possible in a runtime tree (e.g., down to only one hop away from leaf nodes), the checking workload for this tree would be split into very small parts. Each part is equal to checking only one leaf node. In this case, the speedup rate is close to 1 (i.e., almost no faster), as illustrated in Figure 10(b). Other cases are in between.

Therefore, the selected PB splitting node is expected to be as high as possible in a runtime tree. This equals to finding the highest universal or existential node in the tree, and can be done easily. As the number of contexts associated with the selected PB splitting node varies depends on the contexts under checking, the speedup rate is not a constant at runtime. Still, Con-C would never be slower than Seq-C in any case. In practice, due to extra implementation overhead in supporting concurrent checking, the speedup rate may not be as high as analyzed.

5 Evaluation

We implemented Con-C in Java and compared it to Seq-C. For experimental purposes, we made them share the same data structures for representing contexts and consistency constraints. This is for avoiding possible bias in retrieving and manipulating context and constraint data. We built Con-C with the support of Java's java.util.concurrent package, in which method Executors.newCachedThreadPool is used to create a thread pool. The pool helps create new threads for executing tasks as requested, and reuse previously created threads if available.

5.1 Experimental design

In our experiments, we study the following research question: *How efficient is Con-C compared to Seq-C in con-straint checking?* With respect to this question, we decide one dependent variable: *checking time*. It measures how long a given checking task takes for a particular technique. Less checking time implies higher efficiency. With respect to this dependent variable, we decide five independent variables: *checking technique*, *checking workload*, *number of computing units*, *whether to use incremental checking*, and *whether to use balanced checking*. They all affect the dependent variable checking time. We explain them in turn below:

(1) **Checking technique.** We compare Con-C and Seq-C in our experiments.

(2) **Checking workload.** When consistency constraints are fixed for experiments, the checking workload is decided by the contexts under checking. For real-world application scenarios, we cannot control contexts, but we can distinguish different groups of contexts. We would observe how Con-C differs from Seq-C with respect to these different groups of contexts.

(3) **Number of computing units.** This variable decides how much multi-core computing capability can be used for constraint checking. For a machine with n cores, we would control it from 1 to n cores, and observe how Con-C differs from Seq-C in the constraint checking. Note that this variable does not apply to Seq-C.

(4) **Whether to use incremental checking.** Incremental checking means that only those consistency constraints that are affected by context changes are rechecked. Accordingly, non-incremental checking means that all consistency constraints are rechecked whenever any context change occurs. We would compare with incremental checking and with non-incremental checking, how Con-C differs from Seq-C in the constraint checking.

(5) **Whether to use balanced checking.** Our Con-C is automatically balanced. Non-balanced checking is the common idea discussed in the introduction. It assigns the checking of different constraints to different computing units, and these constraints would probably incur different checking workloads (not under control). For ease of controlling and presentation, we allow non-balanced checking as an option of our Con-C. As such, we would compare with balanced checking and with non-balanced checking, how Con-C differs from Seq-C in the constraint checking.

5.2 Experimental subject and setup

We selected a large-scale real-world application SUTPC [7] as our experimental subject. This application uses a central server to collect numerous contexts about taxis in a city, analyzes the city's runtime traffic conditions, and then guides how taxis should head for their destinations to avoid possible traffic jams. In our previous study [7], we have observed an eager need for efficient inconsistency detection for taxi

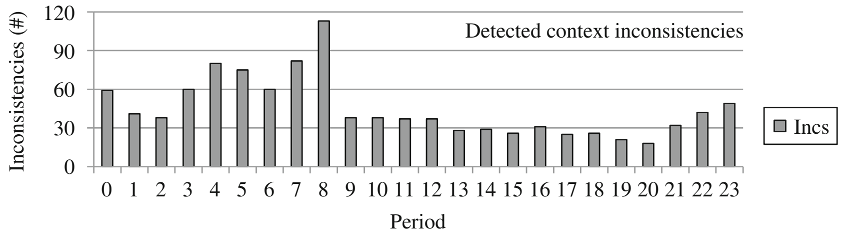

Figure 11 Detected context inconsistencies for 24 groups of taxi contexts.

contexts, as traditional sequential constraint checking techniques could not handle these contexts in time and had to leave about 50% context inconsistencies missed. We previously attempted to address this problem by incremental checking, and in this work we attempted by a concurrent checking approach.

We selected 760 taxis from one company. We monitored these taxis for a continuous 24 h, and obtained a total of 1 545 116 contexts. Each context contains a certain taxi's GPS location, driving speed, driving direction, and service status information at a certain time point. Intervals between every two contexts vary from 20 ms to 3000 ms with an average value of 55.9 ms. This reflects real-world context collection scenarios.

These taxi contexts contain inherent noises. Some contexts seriously deviate from their normal values (e.g., impossibly fast speed or indicating a taxi driving on sea). Some contexts behave as sudden jumps in their physical semantics (e.g., a drastic location change from one place to another remote one in very short time). The application has deployed 12 consistency constraints to verify these contexts' validity and consistency. These constraints can be expressed by our specification language discussed in Subsection 2.2. They totally cover five formula types (universal, "and", "implies", "not", and *bfunc* formulae). Since existential formula is similar to universal formula, and "or" formula is similar to "and"/"implies" formula, these constraints are considered as having sufficiently covered all formula types. Therefore, using them for experimental purposes would not cause bias in the constraint selection.

For comparison purposes, we divided all taxi contexts into 24 groups based on their associated hours, and compared Con-C and Seq-C across all groups. As the total number of contexts is huge (over 1.5 million) and we have to compare Con-C and Seq-C under many configurations (constructed from different value combinations of five independent variables), we chose 5000 contexts from each group for experiments (totally 120 thousand). Considering that we tested more than 12 configurations, we actually checked over 1.44 million contexts, which is close to the original size of 1.5 million. We note that for each group, 5000 contexts were chosen in a random and continual way. This guarantees their properties not altered accidentally. We name these 24 groups of taxi contexts Period 0–23.

All experiments were conducted on a machine with Core i7 CPU at 2.13 GHz and 8 GB RAM. The CPU contains four cores, and were tested with 1, 2, 3, 4 cores, respectively. The operating system on the machine is Microsoft Windows 7 Professional (SP1), and its Java environment is Oracle/Sun JRE 7 update 7.

5.3 Experiment 1: impact of checking workload

We first study the impact of checking workload on the checking efficiency. We compare Seq-C and Con-C under 24 groups of taxi contexts. These groups represent different hours in one day, and exhibit different checking workloads.

Figure 11 illustrates the number of detected context inconsistencies for 24 groups of taxi contexts. We observe that they vary with groups, implying different noisy natures for these different groups of contexts. We note that both Seq-C and Con-C detected the same number of context inconsistencies for all 24 groups. This validates our earlier proved correctness theorem that Con-C always returns the same checking results as Seq-C does.

Figure 12 compares the checking time between Seq-C and Con-C under 24 groups of taxi contexts. We observe that at different hours, the checking workloads do differ, and this affects a particular checking

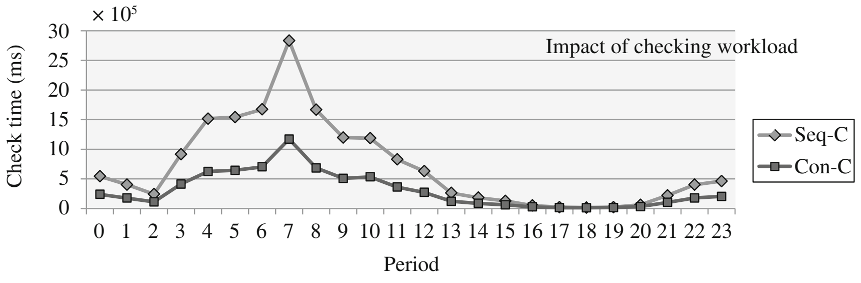

Figure 12 Impact of checking workload (Seq-C vs. Con-C).

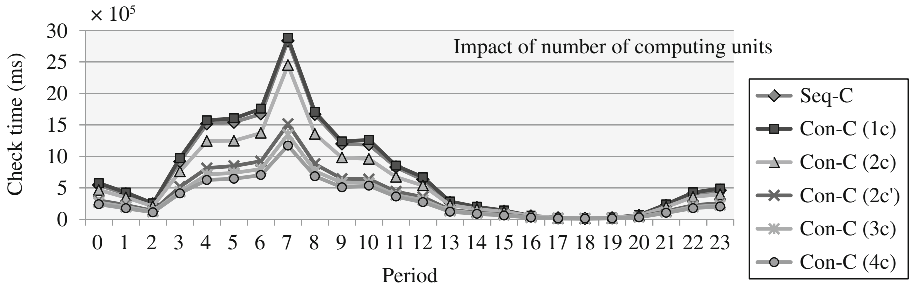

Figure 13 Impact of number of computing units (Seq-C vs. Con-C).

technique's efficiency significantly. However, the trend is that Con-C consistently outperforms Seq-C for all 24 groups of contexts in the checking efficiency. The average ratio (Con-C vs. Seq-C) is 42.9%, i.e., Con-C ran 57.1% faster than Seq-C on average. This result is encouraging.

5.4 Experiment 2: impact of number of computing units

We then study the impact of number of computing units on the checking efficiency. We compare Seq-C and Con-C under 24 groups of taxi contexts, and control the number of computing units (i.e., CPU cores) from 1 to 4. We note two things. First, the number of CPU cores affects Con-C only. Second, for our machine, its CPU contains two physical cores, and the other two cores are simulated by Intel(R) hyper-threading technology. Therefore, when we consider two cores for constraint checking, there are two cases: 1) one physical core and one simulated core; 2) two physical cores. We name these two cases Con-C (2c) and Con-C (2c'), respectively. Note that there is no case of two simulated cores as the first selected core must be a physical one.

Figure 13 compares the checking time between Seq-C and Con-C when the number of CPU cores is controlled from 1 to 4. We observe that with the growth of number of CPU cores, Con-C's checking time consistently reduces more and more for all 24 groups of contexts. When the number of CPU cores is controlled to 1, Con-C's checking time is slightly more than that of Seq-C (4.3% more on average). This is understandable because maintaining the thread pool for concurrent checking inevitably incurs a little bit overhead. With more CPU cores, the ratio between Con-C and Seq-C quickly reduces down to 42.9%. What is worth noticing is that from configuration Con-C (2c) to Con-C (2c'), Con-C also gains new performance. This implies that two physical cores can work really faster than one physical one and one simulated core. This is because the simulated core is essentially based on the same physical core, and they still share some common computing parts in the CPU. On average, when the configuration changes from Con-C (1c), Con-C (2c), Con-C (2c'), Con-C (3c), to Con-C (4c), the ratio between Con-C and Seq-C changes accordingly from 104.3%, 83.8%, 54.7%, 48.2%, to 42.9%. The pace becomes gradually smaller, and this follows the well-known Amdahl's Law.

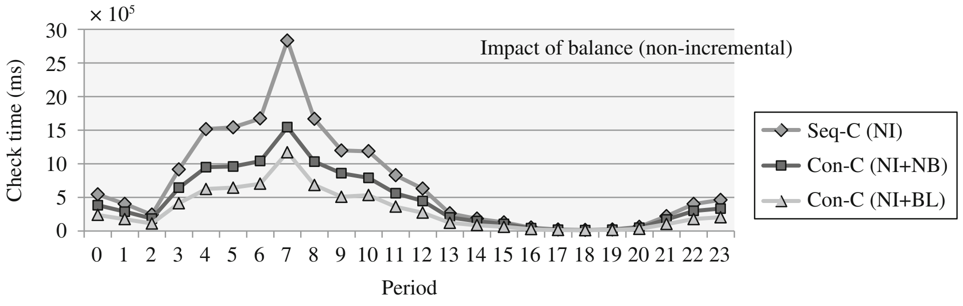

Figure 14 Impact of balance (Seq-C vs. Con-C under non-incremental checking).

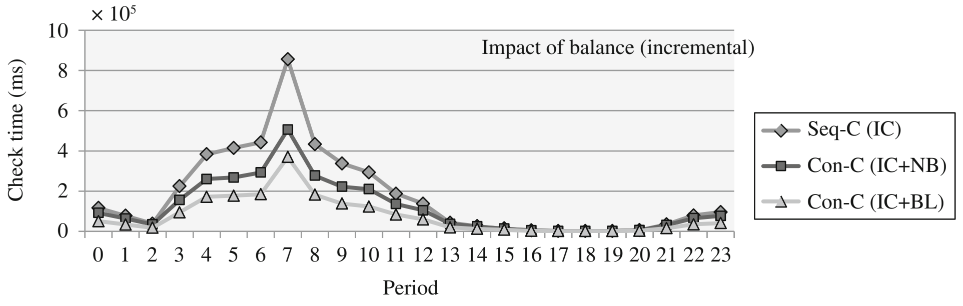

Figure 15 Impact of balance (Seq-C vs. Con-C under incremental checking).

5.5 Experiment 3: impact of incremental checking and balance

We finally study: 1) whether balance is critical to Con-C's checking efficiency, and 2) whether our Con-C is orthogonal to incremental checking. If both answers are yes, then we should be able to observe in experiments that: 1) non-balanced Con-C cannot achieve significant efficiency improvement as balanced Con-C does, and 2) Con-C can additionally improve the checking efficiency based on what has been achieved by incremental checking.

First, we compare Seq-C and Con-C under non-incremental checking. We name them Seq-C (NI) and Con-C (NI), respectively. The latter is further divided into two configurations: non-balanced and balanced. Therefore, we further divide Con-C (NI) into two categories: Con-C (NI+NB) and Con-C (NI+BL). Figure 14 studies the impact of balance on the checking efficiency for these three configurations. We observe that: 1) both Con-C (NI+NB) and Con-C (NI+BL) consistently outperform Seq-C (NI) in the checking efficiency for all 24 groups of taxi contexts; 2) Con-C (NI+BL) additionally outperforms Con-C (NI+NB) in the checking efficiency for all 24 groups of contexts. As compared to Seq-C (NI), the ratio of Con-C (NI+NB) is 65.1% and that of Con-C (NI+BL) is 42.9% on average.

Then, we compare Seq-C and Con-C under incremental checking. Accordingly, we name thus derived three configurations Seq-C (IC), Con-C (IC+NB), and Con-C (IC+BL). They represent Seq-C, non-balanced Con-C, and balanced Con-C under incremental checking, respectively. Figure 15 studies the impact of balance on the checking efficiency for these three configurations. We observe similar trends that: 1) both Con-C (IC+NB) and Con-C (IC+BL) consistently outperform Seq-C (IC) in the checking efficiency for all 24 groups of taxi contexts, although the total time is reduced about one third; 2) Con-C (IC+BL) additionally outperforms Con-C (IC+NB) in the checking efficiency for all 24 groups of contexts. As compared to Seq-C (IC), the ratio of Con-C (IC+NB) is 68.0% and that of Con-C (IC+BL) is 43.0% on average.

From the comparison, we observe that balanced Con-C (our Con-C) can really run faster than non-balanced Con-C (not our Con-C; only a virtually constructed one based on common sense on concurrent checking). In fact, the ratio of non-balanced Con-C ((65.1% + 68.0%) / 2 = 66.6%) with four CPU cores is close to that of balanced Con-C ((83.8% + 54.7%) / 2 = 69.3%) with two CPU cores only. As such, we can conclude that: 1) balance is critical to efficient current checking; 2) our Con-C can gain additional performance bonus (about 57.0%) in addition to what has been obtained by incremental checking (i.e., they complement to each other).

5.6 Summary

From our preceding comparisons between Seq-C and Con-C regarding the impact of checking workload, number of computing units, and incremental checking and balance, our Con-C can conduct constraint checking much more efficiently than Seq-C. Con-C achieves this by automatically exploiting multi-core computing capability. We note that this exploitation is systematically done as our Con-C technique is independent of consistency constraints. Given a consistency constraint, our Con-C semantics automatically analyze its structure and derive PB-splitting nodes for concurrent checking. Our Con-C is thus transparent to constraint developers and application users. This feature is desirable.

6 Related work

In this section, we discuss related work in recent years on addressing context quality problems, detecting inconsistent software artifacts, and resolving detected software inconsistencies.

6.1 Addressing context quality problems

Pervasive computing is receiving increasing attention. Many applications are developed and deployed for pervasive computing environments. To support these applications' context-awareness features, various application frameworks (e.g., Context Toolkit [22], layered architectures [23,24], and adaptable models [25]) and middleware infrastructures (e.g., EgoSpaces [26,27], Lime [28], Gaia [29], Cabot [5,30], and ADAM [31–33]) have been proposed. Most of these research projects have assumed the validity of contexts collected from physical environments, and therefore mainly focused on assisting the development and deployment of context-aware adaptation logics based on these contexts.

However, due to the fact that many applications are actually suffering from low-quality contexts, there is a strong need for these applications to be able to detect their design faults in handling such noisy contexts. Various approaches or techniques have thus been proposed to test or verify context-aware applications. For example, Lu et al. [34,35] presented a new family of test adequacy criteria to cover complex interactions between application logics and their environment contexts. This helps expose those context-related program faults that are otherwise difficult to detect by existing test adequacy criteria. Wang et al. [36] identified context-aware program points in applications, and used them to manipulate test inputs for covering program-environment interactions. This approach fully utilizes existing test cases and enhances them with context-aware features to better expose context-related faults. Sama et al. [37] analyzed common fault patterns in model-based context-aware applications. They proposed exhaustive search to detect these faults using static analysis [38,39]. As static analysis would incur numerous false positives in fault detection, our previous work tried two approaches. One is to stick to static analysis, but we derived a set of deterministic constraints to prune most false positives and a set of likely constraints to rank remaining faults [40]. This attempts to present a list of fault reports to developers, in which top ones are probably true positives (i.e., real faults). The other one is to turn to dynamic analysis, such that all detected faults must be true positives, but we addressed runtime detection efficiency concern using an incremental rule evaluation technique [31–33].

6.2 Detecting inconsistent software artifacts

Besides the work that aims to address the problems caused by noisy contexts in applications, another line of work directly focuses on validating contexts with respect to correctness requirements or consistency constraints. Detecting inconsistencies in contexts somewhat resembles detecting those in other software artifacts. Such software artifacts include UML models [14–16], XML documents [17,18], data structures [19,20], and configuration files [41]. Even particularly for detecting context problems, there is increasingly more work in recent years. Some work focused on detecting anomalies for special types of contexts like RFID contexts [9,10], and some work focused on detecting inconsistencies for general types of contexts [5–7,21].

As analyzed earlier, most inconsistency detection techniques belong to the category of centralized checking. This is intuitive but assumes the availability of all software artifacts under checking to be at one host. Our previous work [21] attempted decentralized checking for mobile devices like smartphones. This protects the privacy of user contexts but trades for this protection by means of extra communication overhead among devices. To address the timeliness requirement for dynamic software artifacts that change quickly or frequently, incremental checking techniques have been studied, e.g., for UML models [15,18,42] and for contexts [6,7]. However, all these techniques have based on single-core machines. We in this paper study concurrent checking for improving the checking efficiency from another dimension. As analyzed and validated by experiments, our Con-C can utilize multi-core computing capability to further improve the checking efficiency. This improvement is orthogonal to that offered by incremental checking, and therefore complements what have been obtained by incremental checking, as we observed in our experiments. Besides, the improvement can be achieved in an automatic way, without requiring constraint developers to do anything.

6.3 Resovling detected software inconsistencies

Detected software inconsistencies need to be resolved such that applications can again obtain a consistent picture of what they aim to handle. Many existing approaches focus on this in terms of different analysis techniques or heuristic strategies. Some of them address particularly context inconsistencies. For example, Bu et al. [43] suggested a very simple yet intuitive strategy of resolving context inconsistencies by discarding all involved contexts until latest ones no longer cause any inconsistency. This is based on the belief that latest contexts always carry newest information required by applications. Insuk et al. [44] considered following human choices in resolving context inconsistencies because they believed that only human beings know what they need. Such choices can be modeled in advance as preference rules to distinguish different priorities in selecting best resolution actions.

Some existing approaches address inconsistencies detected in other software artifacts. For example, Chomicki et al. [45] focused on resolving conflicting actions formed by incoming events captured outside a system. They suggested ignoring an incoming event or randomly discarding several conflicting actions caused by this event. They did so for logical programming, i.e., aiming at logical correctness rather than practical correctness. Ranganathan et al. [29] put it a simple way by setting up rule priorities. This priority information is used to resolve non-deterministic situations where more than one rules are triggered at the same time. Such situations can be considered as a special form of inconsistency as different rules represent different adaptation directions that cannot be taken together. These approaches are mainly based on heuristic strategies or user preferences. They may not be proved as sound but can work in some cases.

We have also tried a heuristic way as well as analytical ones. Our previous work [11] formulated another heuristic rule, aiming to be applicable to more application scenarios. Our previous work [12,13] also attempted to resolve context inconsistencies from an application's perspective, i.e., minimizing potential side effect to the applications that are using these contexts. Besides, we also extended the support of context inconsistency resolution from one application to multiple applications that share the same contexts [46].

Although resolving context inconsistencies take place after these inconsistencies have been detected, they are still closely related to each other. Efficiently detecting context inconsistencies can save valuable time for resolving these inconsistencies as the latter can be even more time-consuming. We in the paper discuss concurrent checking techniques for detecting context inconsistencies, but efficiently resolving these inconsistencies at runtime using multi-core computing capability would also be an interesting research issue.

7 Conclusion

In this paper, we studied the problem of efficiently detecting context inconsistency for Internetware computing. Internetware applications target at open, dynamic environments, and have increasingly more contexts to handle. Detecting noises in these contexts and judging whether these noises would cause contexts inconsistency and affect applications unexpectedly are a critical task and should be done efficiently. The work presented in this paper exactly fits for this need. We proposed our Con-C technique to support concurrent checking. It is based on our analysis of properties of syntax trees and runtime trees associated with consistency constraints under checking. Its correctness and efficiency are formally proved and analyzed.

Unlike the intuitive idea of arbitrarily distributing checking tasks to different computing units, our Con-C always achieves balanced concurrent checking. The balance is achieved at a sub-formula level, and this has a finer granularity than existing work. Besides, this balance is guaranteed to be persistent at runtime, and obtaining this persistent balance is totally transparent to constraint developers and application users. That is, they do not have to make any parameter tuning for Con-C so as to fully utilize multi-core computing capability. This is desirable. We evaluated Con-C with a real-world context-aware application that uses huge volumes of dynamic contexts. We believe that such a practical evaluation can more realistically validate the comparison between our Con-C and existing techniques that check contexts sequentially. The evaluation results confirmed that Con-C can bring significant additional gains in the checking efficiency.

Our Con-C complements many existing incremental techniques. It achieves performance gains in addition to what have been obtained by incremental checking, as our evaluation shows. This indicates that Our Con-C and existing incremental checking can work together to realize an amazing improvement in the checking efficiency. Still, we note that this complementation works at different levels, i.e., incremental checking works at a constraint level and our Con-C works at a sub-formula level. This avoids their possible conflicts, and this also explains why they can complement each other. For incremental checking that also works at a sub-formula level, e.g., our previous partial constraint checking technique [7], our Con-C may not work so efficiently (but can be still faster), as its balance may not be persistent when tree structures are changed by partial checking. We are investigating this issue, and plan to extend our concurrent checking idea to more techniques as well as more application scenarios. This would solidly support Internetware applications when they handle more and more uncontrolled noisy contexts generated from open, dynamic environments.

Finally, although this work mainly focuses on performance improvement for consistency checking, the improved performance can also contribute to the quality of detected context inconsistencies, according to our previous study [7]. This behaves as more context inconsistencies being detected in time (less inconsistencies being missed). Therefore, a comprehensive study on how much additional effectiveness rather than efficiency can be achieved by such concurrent checking is an interesting issue and deserves investigation. We also plan to make it our future work.

Acknowledgements

This research was partially funded by National Basic Research Program of China (973 Program) (Grant No. 2009CB320702), National High-tech R&D Program (863 Program) (Grant No. 2012AA011205), National Natural Science Foundation of China (Grant Nos. 61100038, 61021062), and Research Grants Council (612210) of Hong Kong. Chang Xu was also partially supported by Program for New Century Excellent Talents in University, China (Grant No. NCET-10-0486). We sincerely thank this special issue's editors and anonymous reviewers for their valuable suggestions and comments for improving this paper.

References

1 Lv J, Ma X, Hunag Y, et al. Internetware: a shift of software paradigm. In: Proceedings of the 1st Asia-Pacific Symposium on Internetware, Beijing, 2009. 1–9

2 Lv J, Ma X, Tao X, et al. Internetware-oriented environmental driving models and supporting technology (in Chinese). Sci China Ser F-Inf Sci, 2008, 38: 864–900

3 Lv J, Ma X, Tao X, et al. Research progress on Internetware (in Chinese). Sci China Ser F-Inf Sci, 2006, 36: 1037–1080

4 Lv J, Ma X, Tao X, et al. Explicit environmental constructs for Internetware (in Chinese). Sci Sinica Inf Sci, 2013, 43: 1–23

5 Xu C, Cheung S C. Inconsistency detection and resolution for context-aware middleware support. In: Proceedings of the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Lisbon, 2005. 336–345

6 Xu C, Cheung S C, Chan W K. Incremental consistency checking for pervasive context. In: Proceedings of the 28th International Conference on Software Engineering, Shanghai, 2006. 292–301

7 Xu C, Cheung S C, Chan W K, et al. Partial constraint checking for context consistency. ACM Trans Softw Eng Methodol, 2010, 19: 1–61

8 Garfinkel S, Rosenberg B. RFID: Applications, Security, and Privacy. Addison-Wesley, 2005

9 Jeffery S R, Garofalakis M, Frankin M J. Adaptive cleaning for RFID data streams. In: Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, 2006. 163–174

10 Rao J, Doraiswamy S, Thakkar H, et al. A deferred cleansing method for RFID data analytics. In: Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, 2006. 175–186

11 Xu C, Cheung S C, Chan W K, et al. Heuristics-based strategies for resolving context inconsistencies in pervasive computing applications. In: Proceedings of the 28th International Conference on Distributed Computing Systems, Beijing, 2008. 713–721

12 Xu C, Cheung S C, Chan W K, et al. 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 Symposium on the Foundations of Software Engineering, Dubrovnik, 2007. 569–572

13 Xu C, Ma X, Cao C, et al. Minimizing the side effect of context inconsistency resolution for ubiquitous computing. In: Proceedings of the 8th ICST International Conference on Mobile and Ubiquitous Systems, LNICST 104, Copenhagen, 2011. 285–297

14 Egyed A. Instant consistency checking for the UML. In: Proceedings of the 28th International Conference on Software Engineering, Shanghai, 2006. 381–390

15 Reiss S P. Incremental maintenance of software artifacts. IEEE Trans Softw Eng, 2006, 32: 682–697

16 Xiong Y, Hu Z, Zhao H, et al. Supporting automatic model inconsistency fixing. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Amsterdam, 2009. 315–324

17 Nentwich C, Capra L, Emmerich W, et al. xlinkit: a consistency checking and smart link generation service. ACM Trans Internet Technol, 2002, 2: 151–185

18 Nentwich C, Emmerich W, Finkelstein A, et al. Flexible consistency checking. ACM Trans Softw Eng Methodol, 2003, 12: 28–63

19 Demsky B, Rinard M. Data structure repair using goal-directed reasoning. In: Proceedings of the 27th International Conference on Software Engineering, St. Louis, 2005. 176–185

20 Demsky B, Rinard M. Goal-directed reasoning for specification-based data structure repair. IEEE Trans Softw Eng, 2006, 32: 931–951

21 Xu C, Cheung S C. Decentralized constraint checking for pervasive computing. In: Proceedings of the 6th Annual IEEE International Conference on Pervasive Computing and Communications (Ph.D. forum), Hong Kong, 2008. 45–48

22 Salber D, Dey A K, Abowd G D. The context toolkit: aiding the development of context-enabled applications. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, Pittsburgh, 1999. 434–441

23 Griswold W G, Boyer R, Brown S W, et al. A component architecture for an extensible, highly integrated context-aware computing infrastructure. In: Proceedings of the 25th International Conference on Software Engineering, Portland, 2003. 363–372

24 Henricksen K, Indulska J. A software engineering framework for context-aware pervasive computing. In: Proceedings the 2nd IEEE Conference on Pervasive Computing and Communications, Orlando, 2004. 77–86

25 Zachariadis S, Mascolo C, Emmerich W. The SATIN component system—a metamodel for engineering adaptable mobile systems. IEEE Trans Softw Eng, 2006, 32: 910–927

26 Julien C, Roman G C. Egocentric context-aware programming in ad hoc mobile environments. In: Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering, Charleston, 2002. 21–30

27 Julien C, Roman G C. EgoSpaces: facilitating rapid development of context-aware mobile applications. IEEE Trans Softw Eng, 2006, 32: 281–298

28 Murphy A L, Picco G P, Roman G C. LIME: a coordination model and middleware supporting mobility of hosts and agents. ACM Trans Softw Eng Methodol, 2006, 15: 279–328

29 Ranganathan A, Campbell R H. An infrastructure for context-awareness based on first order logic. Pers Ubiquitous

Comput, 2003, 7: 353–364
30 Xu C, Cheung S C, Lo C, et al. Cabot: on the ontology for the middleware support of context-aware pervasive applications. In: Proceedings of the NPC 2004 Workshop on Building Intelligent Sensor Networks, Wuhan, 2004. 568–575
31 Xu C, Cheung S C, Ma X, et al. ADAM: identifying defects in context-aware adaptation. J Syst Softw, 2012, 85: 2812–2828
32 Xu C, Cheung S C, Ma X, et al. Dynamic fault detection in context-aware adaptation. In: Proceedings of the 4th Asia-Pacific Symposium on Internetware, Qingdao, 2012. 1–10
33 Xu C, Cheung S C, Ma X, et al. Detecting faults in context-aware adaptation. Int J Softw Inf, 2013, 7: 85–111
34 Lu H, Chan W, Tse T. Testing context-aware middleware-centric programs: a data flow approach and a RFID-based experimentation. In: Proceedings of the 14th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Portland, 2006. 242–252
35 Lu H, Chan W, Tse T. Testing pervasive software in the presence of context inconsistency resolution services. In: Proceedings of the 30th International Conference on Software Engineering, Leipzig, 2008. 61–70
36 Wang Z, Elbaum S, Rosenblum D S. Automated generation of context-aware tests. In: Proceedings of the 29th International Conference on Software Engineering, Minneapolis, 2007. 406–415
37 Sama M, Rosenblum D S, Wang Z, et al. Multi-layer faults in the architectures of mobile, context-aware adaptive applications. J Syst Softw, 2010, 83: 906–914
38 Sama M, Elbaum S, Raimondi F, et al. Context-aware adaptive applications: fault patterns and their automated identification. IEEE Trans Softw Eng, 2010, 36: 644–661
39 Sama M, Rosenblum D S, Wang Z, et al. Model-based fault detection in context-aware adaptive applications. In: Proceedings of the 16th ACM SIGSOFT International Symposium on the Foundations of Software Engineering, Atlanta, 2008. 261–271
40 Liu Y, Xu C, Cheung S C. AFChecker: effective model checking for context-aware adaptive applications. J Syst Softw, 2013, 86: 854–867
41 Xiong Y, Hubaux A, She S, et al. Generating range fixes for software configuration. In: Proceedings of the 34th International Conference on Software Engineering, Zurich, 2012. 58–68
42 Falleri J R, Blanc X, Bendraou R, et al. Incremental inconsistency detection with low memory overhead. Softw Pract Exper, 2012, doi: 10.1002/spe.2171
43 Bu Y, Gu T, Tao X, et al. Managing quality of context in pervasive computing. In: Proceedings of the 6th International Conference on Quality Software, Beijing, 2006. 193–200
44 Insuk P, Lee D, Hyun S J. A dynamic context-conflict management scheme for group-aware ubiquitous computing environments. In: Proceedings of the 29th Annual International Computer Software and Applications Conference, Edinburgh, 2005. 359–364
45 Chomicki J, Lobo J, Naqvi S. Conflict resolution using logic programming. IEEE Trans Knowl Data Eng, 2003, 15: 244–249
46 Yang H, Xu C, Ma X, et al. ConsView: towards application-specific consistent context views. In: Proceedings of the 36th Annual International Computer Software and Applications Conference, Izmir, 2012. 632–637