Minimizing Link Generation in Constraint Checking for Context Inconsistency Detection

Chuyang Chen[†‡], Huiyan Wang*[†‡], Lingyu Zhang[†‡], 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
chuyangchen2018@outlook.com, why@nju.edu.cn, zly@smail.nju.edu.cn, {changxu, yuping}@nju.edu.cn

***Abstract*—Adaptive applications rely on conditions about their environments (or *contexts*) to deliver smart services, e.g., location-aware services. Due to inherent noises in environmental sensing and interpretation, there is an increasing demand for guarding the consistency of contexts to avoid application misbehavior, and at the same time minimizing the guarding cost. Existing work has tried to reduce the cost by speeding up the kernel constraint checking module inside the consistency guarding process. Most efforts have been spent on reusing previous checking results or checking constraints in parallel, while leaving untouched one central problem of *link generation*, the step that consumes a substantially large part of the total time cost for explaining why constraints have been violated. In this paper, we propose a novel technique, MG, to automatically identify and remove redundant link generation, without harming any checking result. We show that MG is *sound* (always checking correctly) and *complete* (removing all redundancy). Our experiments with synthesized and real-world consistency constraints reported that compared with existing work, MG achieved significant efficiency improvements on the link generation (tens to hundreds times speedup), and could reduce the total constraint checking time up to 45.4%.**

***Index Terms*—Context inconsistency, constraint checking, link redundancy**

I. Introduction

With the advances of modern sensor and actuator technologies, adaptive applications (e.g., location-aware navigators, self-driving vehicles [1], [2], cloud computing systems [3], and mobile apps [4]) are gaining increasing popularity. These applications deliver intelligent services by interpreting their environments as *contexts* [5] and based on them conducting adaptive behaviors. However, due to inevitable environmental noises, application contexts can easily deviate from their ground truths (e.g., inaccurate location sensing) [6]–[10], and thus lead to application misbehaviors or even crashes.

Due to the lack of direct ground truths, various approaches have studied ways of detecting flaws in application contexts and explaining why they have occurred. One promising approach is to check an application's contexts against pre-specified rules, namely, *consistency constraints* [6]–[8], that should hold under the application's knowledge domain and physical laws (e.g., a continual change of location data should not exceed the speed limit). Then, any detection of constraint violation is considered as a *context inconsistency* [6]–[8], and reported to the application for resolution [11]–[17]. As the constraint checking process aims to guard an application's reliability, it is expected to be *effective* and *efficient* [7], [8], [18]–[21], without compromising the application's normal functionalities. To this end, various techniques have studied speeding up constraint checking, e.g., entire constraint checking (ECC) [6] as the correctness baseline, partial constraint checking (PCC) [8] to check constraints incrementally by reusing previous results, and concurrent constraint checking (Con-C) [18] and GPU-assisted concurrent checking (GAIN) [19] by checking constraints in parallel via multiple CPU or GPU threads.

With such efforts, further performance improvement has become extremely difficult for constraint checking. Considering the growing environmental complexity and dynamics (e.g., ubiquitous cyber-physical interactions and huge-volume data) [22], the increasing workload has breached the capabilities of existing constraint checking techniques (e.g., over 90% missed inconsistency detection [20]).

In this paper, we tackle this problem from a new perspective. We characterize existing efforts (e.g., incremental or concurrent constraint checking) into the "making-it-faster" category, i.e., speeding up all calculations in constraint checking. We conjecture a new "making-it-less" category of efforts, i.e., identifying and removing redundant calculations in constraint checking, without affecting any result. If the conjecture can be realized, not only it itself can reduce the overhead of constraint checking, but also it can assist all existing techniques, contributing generally to their further improvements.

We dig into constraint checking, and observe a two-step pattern that covers all calculations in existing constraint checking techniques, namely, *truth value evaluation* and *link generation*. The former examines whether a given consistency constraint is violated with respect to the contexts under checking, and returns a truth value of True or False. The latter generates a data structure named *link* [6]–[8] to explain which elements in the contexts have concretely contributed to the constraint violation (when False) or satisfaction (when True), helping developers to locate malfunctioned places, and it can consume a substantially large part (could be up to 45% as reported by our experiments) in the total checking cost. Besides, we observe that many links generated as intermediate results during constraint checking are essentially *redundant* (23–100%, as reported), in the sense that being without them never affects generating final link results (details analyzed later). Then, how such redundant links have been generated

*Corresponding authors.

and whether one can avoid them become both an interesting question and the key towards our “making-it-less” conjecture. In this paper, we study this problem and aim to eliminate such link redundancy completely.

We propose a Minimized Link Generation technique (or MG) to automatically identify and remove redundant link generation during constraint checking. As compared to existing practice of link generation, named Complete Link Generation (or CG), working as the baseline and used in existing constraint checking techniques [6], [8], [18], MG can significantly reduce link generation by 23–100%. MG achieves this via a hybrid static-dynamic analysis, first constructing a data structure named S-CCT encoding a constraint’s static syntax information, and then evolving it with dynamic truth value information associated with the contexts checked on this constraint. We prove that the S-CCT has marked all substantial places that are necessary for generating the final links to explain the constraint’s violation or satisfaction, and that other places can be safely isolated from consideration, without any chance to affect the calculation of the final links (*soundness*). We also show that MG is *complete* in having identified and removed all redundant link generation, and generic in being applicable to all existing constraint checking techniques.

Our experimental evaluation reported that: (1) MG realized 100% link utilization (i.e., removing 100% redundant link generation), as compared to existing work, which could lead to severe link redundancy problem (e.g., 75–83% average link redundancy); (2) When applied to existing constraint checking techniques (e.g., ECC [6], PCC [8], and Con-C [18]), MG brought significant efficiency improvements on the link generation (tens or hundreds of improvements), and reduced the total constraint checking up to 45.4%. The evaluation confirmed MG’s generic benefits, by boosting existing constraint checking techniques towards further improvements.

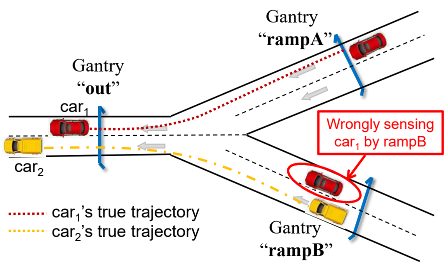
The remainder of this paper is organized as follows. Section II uses an illustrative example to introduce the background. Section III first reports a pilot study to motivate our work, and then elaborates on our MG technique to identify and eliminate redundant link generation in constraint checking. Section IV evaluates our MG under controlled experiments with exhaustive constraint analysis and a case study with real-world scenarios. Section V discusses the related work in recent years, and finally Section VI concludes this paper.

II. Background and Motivation

In this section, we introduce preliminary concepts with an illustrative example.

A. Preliminary

Context and context pool. A *context* is a piece of structured information about an application’s running environment [20]. It can be modeled as a finite set of elements, each denoting a relevant part of this context. Consider a highway charging system example, which tracks each travelling vehicle’s information like license plate number, driving speed, and current location, and calculates related highway tolls. One



Gantry
“rampA”
Gantry
“out”
car$_1$
car$_2$
Wrongly sensing car$_1$ by rampB
car$_1$’s true trajectory
car$_2$’s true trajectory
Gantry
“rampB”

Fig. 1: Illustration of a highway scenario

can model the vehicles that have recently driven through a specific highway gantry a as: $\mathrm{C}_{\mathsf{a}} = \{\mathsf{car}_1, \mathsf{car}_2, \cdots\}$. Each car_i identifies a vehicle with specific information, e.g., car id, gantry number, driving speed, and timestamp. When the time flows, contexts are updated to meet the application’s requirements. A *context pool* collects all contexts interesting to an application, e.g., the preceding contexts associated with all highway gantries: C_{a}, C_{b}, ..., C_{m}.

Consistency constraint. For the preceding application, highway gantries deploy cameras and sensors to track vehicles, and the tracking can be subject to recognition or sensing error. As a result, thus maintained contexts can be inaccurate, incomplete, or even conflict with each other, which are known as *consistency constraints* [6]–[8]. To address this, *consistency constraints* can be formulated to check problems with the maintained contexts. We give an example constraint R_{exit} below:

$$\begin{aligned}\forall v_1 \in \mathrm{C}_{\text{out}}\big((\exists v_2 \in \mathrm{C}_{\text{rampA}}\,(\mathrm{sameCar}\,(v_1, v_2)))\\ \quad \mathsf{implies}\ (\mathsf{not}\ (\exists v_3 \in \mathrm{C}_{\text{rampB}}\,(\mathrm{sameCar}\,(v_1, v_3))))\big)\end{aligned} \quad (\mathrm{R}_{\text{exit}})$$

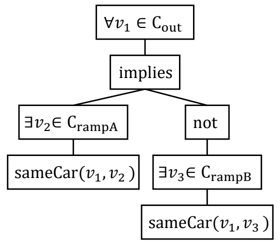
This constraint requests that any exiting vehicle at Gantry out should be from either Gantry rampA or rampB, but not both. In particular, if the exiting vehicle has been detected at Gantry rampA, it should not appear at Gantry rampB (Fig. 1 illustrates a possible sensing error). Otherwise, its highway toll may be calculated wrongly.

Such consistency constraints are specified using the following first-order logical formulae (used in existing work [6]–[8], [18], [20], [23]–[25]), where C is a context, v_i is a variable that takes an element from C as its value; terminal *bfunc* is an application-specific predicate that returns a Boolean value (True/T or False/F):

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

A consistency constraint can also be represented by a syntax tree structure, e.g., R_{exit} illustrated in Fig. 2.

Constraint checking. *Constraint checking* examines the contexts in an application’s context pool against given consistency constraints to report *truth values* (indicating whether constraints are violated or satisfied) and *links* (indicating how constraints are violated or satisfied). This corresponds



$\forall v_1 \in C_{out}$
implies
$\exists v_2 \in C_{rampA}$
not
sameCar(v_1, v_2)
$\exists v_3 \in C_{rampB}$
sameCar(v_1, v_3)

Fig. 2: Syntax tree structure of constraint R_{exit}

$$\mathcal{T}[\forall v \in \mathrm{C}(f)]_\alpha = \mathrm{T} \wedge \left(\bigwedge_{\mathrm{e} \in \mathrm{C}} \mathcal{T}[f]_{\alpha[v:=\mathrm{e}]} \right)$$
$$\mathcal{T}[\exists v \in \mathrm{C}(f)]_\alpha = \mathrm{F} \vee \left(\bigvee_{\mathrm{e} \in \mathrm{C}} \mathcal{T}[f]_{\alpha[v:=\mathrm{e}]} \right)$$
$$\mathcal{T}\left[(f_1) \ \ \mathsf{implies} \ \ (f_2)\right]_\alpha = \neg\mathcal{T}\left[f_1\right]_\alpha \vee \mathcal{T}\left[f_2\right]_\alpha$$
$$\mathcal{T}\left[\mathsf{not} \ \ (f)\right]_\alpha = \neg\mathcal{T}\left[f\right]_\alpha$$
$$\mathcal{T}\left[bfunc(v_1, \ldots, v_n)\right]_\alpha = bfunc(v_1, \ldots, v_n)_\alpha$$

Fig. 3: Part of semantics for the truth value evaluation

to two kernel steps, namely, *truth value evaluation* and *link generation*. We introduce them below.

B. Constraint Checking

Consider the preceding consistency constraint R_{exit}. Let $C_{\text{out}} = \{\mathsf{car}_1, \mathsf{car}_2\}$, $C_{\text{rampA}} = \{\mathsf{car}_1\}$, and $C_{\text{rampB}} = \{\mathsf{car}_1, \mathsf{car}_2\}$. Then the constraint is violated, resulting in a truth value of False (F), along with a link of $\langle \mathsf{violated}, \{v_1 = \mathsf{car}_1, v_2 = \mathsf{car}_1, v_3 = \mathsf{car}_1\}\rangle$, explaining that element car_1 in contexts C_{out}, C_{rampA}, and C_{rampB} together decide the constraint's violation (while other elements are irrelevant).

To discuss how to obtain the above truth value and link, we let $f_A = \exists v_2 \in \mathrm{C}_{\text{rampA}} \left(\mathrm{sameCar}\left(v_1, v_2\right)\right)$ and $f_B = \exists v_3 \in \mathrm{C}_{\text{rampB}} \left(\mathrm{sameCar}\left(v_1, v_3\right)\right)$, for ease of presentation. Then, the proceeding constraint can be simplified as: $\mathrm{R}_{\text{exit}} = \forall v_1 \in \mathrm{C}_{\text{out}}\big((f_A) \, \mathsf{implies} \, (\mathsf{not} \, (f_B))\big)$.

1) Truth Value Evaluation: To conduct the truth value evaluation, constraint R_{exit} would be evaluated to a Boolean value according to its contained contexts. Fig. 3 lists part of semantics of the truth value evaluation existing constraint checking techniques follow (for example only, but already sufficient for evaluating R_{exit}). $\mathcal{T}[f]_\alpha$ denotes the evaluation of formula f under variable assignment α. The semantics are defined recursively and can be used accordingly.

To evaluate R_{exit}, one needs to enumerate all possible variable assignments for the top universal formula's subformula (the $\mathsf{implies}$ one):

$$\begin{aligned}
&\mathcal{T}\left[\forall v_1 \in \mathrm{C}_{\text{out}}\big((f_A) \, \mathsf{implies} \, (\mathsf{not} \, (f_B))\big)\right]_\emptyset \\
=&\mathrm{T} \wedge \mathcal{T}\left[(f_A) \, \mathsf{implies} \, (\mathsf{not} \, (f_B))\right]_{\langle v_1 := \mathsf{car}_1\rangle} \\
&\wedge \mathcal{T}\left[(f_A) \, \mathsf{implies} \, (\mathsf{not} \, (f_B))\right]_{\langle v_1 := \mathsf{car}_2\rangle} .
\end{aligned}$$

We explain the calculation of the first variable assignment (i.e., $v_1 := \mathsf{car}_1$) for example:

$$\begin{aligned}
&\mathcal{T}\left[(f_A) \, \mathsf{implies} \, (\mathsf{not} \, (f_B))\right]_{\langle v_1 := \mathsf{car}_1\rangle} \\
=&\neg\mathcal{T}\left[(f_A)\right]_{\langle v_1 := \mathsf{car}_1\rangle} \vee \mathcal{T}\left[(\mathsf{not} \, (f_B))\right]_{\langle v_1 := \mathsf{car}_1\rangle} \\
=&\neg\mathcal{T}\left[(f_A)\right]_{\langle v_1 := \mathsf{car}_1\rangle} \vee \neg\mathcal{T}\left[(f_B)\right]_{\langle v_1 := \mathsf{car}_1\rangle} .
\end{aligned}$$

We then follow other semantics to evaluate f_A and f_B:

$$\begin{aligned}
&\mathcal{T}\left[(f_A)\right]_{\langle v_1 := \mathsf{car}_1\rangle} \\
=&\mathcal{T}\left[\exists v_2 \in \mathrm{C}_{\text{rampA}} \left(\mathrm{sameCar}\left(v_1, v_2\right)\right)\right]_{\langle v_1 := \mathsf{car}_1\rangle} \\
=&\mathrm{F} \vee \mathcal{T}\left[\mathrm{sameCar}\left(v_1, v_2\right)\right]_{\langle v_1 := \mathsf{car}_1, v_2 := \mathsf{car}_1\rangle} = \mathrm{T},
\end{aligned}$$

$$\begin{aligned}
&\mathcal{T}\left[(f_B)\right]_{\langle v_1 := \mathsf{car}_1\rangle} \\
=&\mathcal{T}\left[\exists v_3 \in \mathrm{C}_{\text{rampB}} \left(\mathrm{sameCar}\left(v_1, v_3\right)\right)\right]_{\langle v_1 := \mathsf{car}_1\rangle} \\
=&\mathrm{F} \vee \mathcal{T}\left[\mathrm{sameCar}\left(v_1, v_3\right)\right]_{\langle v_1 := \mathsf{car}_1, v_3 := \mathsf{car}_1\rangle} \\
&\vee \mathcal{T}\left[\mathrm{sameCar}\left(v_1, v_3\right)\right]_{\langle v_1 := \mathsf{car}_1, v_3 := \mathsf{car}_2\rangle} \\
=&\mathrm{F} \vee \mathrm{T} \vee \mathrm{F} = \mathrm{T}.
\end{aligned}$$

With them, we obtain $\mathcal{T}\left[(f_A) \, \mathsf{implies} \, (\mathsf{not} \, (f_B))\right]_{\langle v_1 := \mathsf{car}_1\rangle} = \mathrm{F}$. Similarly, we obtain $\mathcal{T}\left[(f_A) \, \mathsf{implies} \, (\mathsf{not} \, (f_B))\right]_{\langle v_1 := \mathsf{car}_2\rangle} = \mathrm{T}$. Then we obtain the whole constraint's truth value:

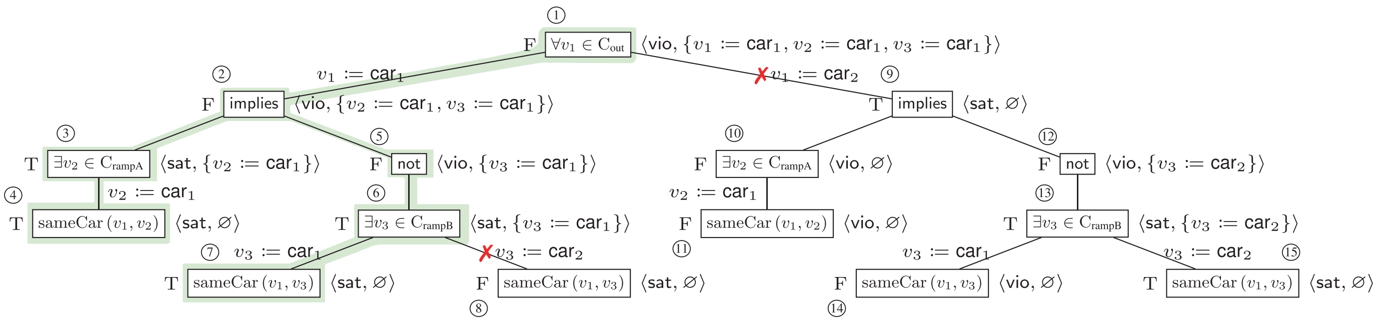
$$\mathcal{T}\left[\forall v_1 \in \mathrm{C}_{\text{out}}\big((f_A) \, \mathsf{implies} \, (\mathsf{not} \, (f_B))\big)\right] = \mathrm{T} \wedge \mathrm{F} \wedge \mathrm{T} = \mathrm{F}.$$

We note that the above calculation has been explained in a top-down manner for human understanding. The truth value evaluation is actually conducted in a bottom-up manner, i.e., a post-order traversal: when all necessary nodes are evaluated, their parent nodes can then be evaluated. One can also annotate the constraint's syntax tree structure with variable assignments to expand it for better illustration, as in Fig. 4, which is named *consistency computation tree* (or *CCT*) as proposed in existing work [7], [8], [18]–[20], [23], [25]). We also list the truth values evaluated for all nodes for reference in Fig. 4.

2) Link Generation: With truth values obtained, the link generation can next be conducted to explain why the associated violations (when F) or satisfaction (when T) have occurred via links. For the preceding example, its generation results contain one link: $\langle \mathsf{violated}, \, \{v_1 = \mathsf{car}_1, \, v_2 = \mathsf{car}_1, \, v_3 = \mathsf{car}_1\}\rangle$, which can be calculated by following the semantics in Fig. 5 (part but sufficient for the example). In the link, the first part indicates the constraint violation, and the second part includes the certain variable assignment that leads to this violation. In the semantics in Fig. 5, $\mathcal{L}[f]_\alpha$ denotes generating links for formula f under variable assignment α. There are some auxiliary functions or operations, e.g., FlipSet is used to flip the first part for all links in a given set (i.e., from violated to satisfied, and vice versa), and $\otimes$ is used to merge two link sets by the Cartesian product $(\cdot)$ upon each pair formed between a link from the first set and another from the second set. For example, two link sets S_1 and S_2 are merged as follows:

$$S_1 \otimes S_2 = \{l_1 \cdot l_2 | l_1 \in S_1, l_2 \in S_2\} .$$

where $l_1 \cdot l_2 = \langle Type(l_1), Bindings(l_1) \cup Bindings(l_2)\rangle$, with $Type(l)$ referring l's link type (same for l_1 and l_2 when



①
F $\forall v_1 \in \mathrm{C_{out}}$ ⟨vio, {v_1 := car$_1$, v_2 := car$_1$, v_3 := car$_1$}⟩
② v_1 := car$_1$
F implies ⟨vio, {v_2 := car$_1$, v_3 := car$_1$}⟩
③ T $\exists v_2 \in \mathrm{C_{rampA}}$ ⟨sat, {v_2 := car$_1$}⟩
④ v_2 := car$_1$
T sameCar (v_1, v_2) ⟨sat, ∅⟩
⑤ F not ⟨vio, {v_3 := car$_1$}⟩
⑥ T $\exists v_3 \in \mathrm{C_{rampB}}$ ⟨sat, {v_3 := car$_1$}⟩
⑦ v_3 := car$_1$
T sameCar (v_1, v_3) ⟨sat, ∅⟩
v_3 := car$_2$
F sameCar (v_1, v_3) ⟨sat, ∅⟩
⑧
v_1 := car$_2$ ⑨
T implies ⟨sat, ∅⟩
⑩ F $\exists v_2 \in \mathrm{C_{rampA}}$ ⟨vio, ∅⟩
v_2 := car$_1$
F sameCar (v_1, v_2) ⟨vio, ∅⟩
⑪
⑫ F not ⟨vio, {v_3 := car$_2$}⟩
⑬ T $\exists v_3 \in \mathrm{C_{rampB}}$ ⟨sat, {v_3 := car$_2$}⟩
v_3 := car$_1$
F sameCar (v_1, v_3) ⟨vio, ∅⟩
⑭
v_3 := car$_2$ ⑮
T sameCar (v_1, v_3) ⟨sat, ∅⟩

Fig. 4: CCT for the example constraint (left: truth value; right: link; shadowed sub-tree: S-CCT part)

$$\mathcal{L}\left[\forall v \in \mathrm{C}(f)\right]_\alpha = \{\langle \mathsf{vio}, \{v := \mathrm{e}\}\rangle \otimes \mathcal{L}[f]_{\alpha[v:=\mathrm{e}]} \mid \mathrm{e} \in \mathrm{C} \wedge \mathcal{T}[f]_{\alpha[v:=\mathrm{e}]} = \mathrm{F}\}$$

$$\mathcal{L}\left[\exists v \in \mathrm{C}(f)\right]_\alpha = \{\langle \mathsf{sat}, \{v := \mathrm{e}\}\rangle \otimes \mathcal{L}[f]_{\alpha[v:=\mathrm{e}]} \mid \mathrm{e} \in \mathrm{C} \wedge \mathcal{T}[f]_{\alpha[v:=\mathrm{e}]} = \mathrm{T}\}$$

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

$$\begin{aligned}
&(1) && \mathsf{FlipSet}\left(\mathcal{L}\left[f_1\right]_\alpha\right) \otimes \mathcal{L}\left[f_2\right]_\alpha, \text{ if } \mathcal{T}\left[f_1\right]_\alpha = \mathrm{T} \text{ and } \mathcal{T}\left[f_2\right]_\alpha = \mathrm{F} \\
&(2) && \mathsf{FlipSet}\left(\mathcal{L}\left[f_1\right]_\alpha\right) \cup \mathcal{L}\left[f_2\right]_\alpha, \text{ if } \mathcal{T}\left[f_1\right]_\alpha = \mathrm{F} \text{ and } \mathcal{T}\left[f_2\right]_\alpha = \mathrm{T} \\
&(3) && \mathcal{L}\left[f_2\right]_\alpha, \text{ if } \mathcal{T}\left[f_1\right]_\alpha = \mathcal{T}\left[f_2\right]_\alpha = \mathrm{T} \\
&(4) && \mathsf{FlipSet}\left(\mathcal{L}\left[f_1\right]_\alpha\right), \text{ if } \mathcal{T}\left[f_1\right]_\alpha = \mathcal{T}\left[f_2\right]_\alpha = \mathrm{F}
\end{aligned}$$

$$\mathcal{L}\left[\text{not } (f)\right]_\alpha = \mathsf{FlipSet}\left(\mathcal{L}[f]_\alpha\right)$$

$$\mathcal{L}\left[bfunc(v_1, \ldots, v_n)\right]_\alpha =$$

$$\begin{aligned}
&(1) && \{\langle \mathsf{sat}, \varnothing\rangle\}, \text{ if } \mathcal{T}\left[bfunc(v_1, \ldots, v_n)\right]_\alpha = \mathrm{T} \\
&(2) && \{\langle \mathsf{vio}, \varnothing\rangle\}, \text{ if } \mathcal{T}\left[bfunc(v_1, \ldots, v_n)\right]_\alpha = \mathrm{F}
\end{aligned}$$

Fig. 5: Part of semantics for the link generation ("vio" stands for "violated" and "sat" stands for "satisfied")

used), and $Bindings(l)$ referring to l's contained variable-value bindings in its variable assignment.

Similarly, the link generation is also conducted in a bottom-up manner, i.e., when all necessary nodes obtain links, their parent nodes obtain too. One can follow the semantics in Fig. 5 to calculate all links (intermediate and final ones) for the constraint. For illustration, we annotate all such links on the CCT associated with this constraint in Fig. 4.

We observe that for calculating the final link at node 1 in Fig. 4, all nodes on the tree have participated into the calculation and generated their own links. As observed from the semantics in Fig. 3 and Fig. 5, the link generation is much more complicated than the truth value evaluation. On the other hand, some generated links are indeed not necessary. For example, we observe that the final link at node 0 essentially relates to links at nodes 1–7 only, i.e., those at nodes 8–15 are redundant. The redundancy rate (53.3%) is rather high!

III. Methodology

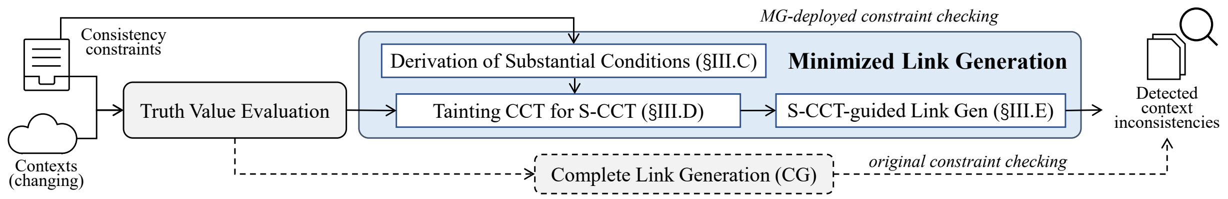
In this section, we first report a pilot study to motivate our work, and then elaborate on our MG methodology of eliminating redundant link generation, to realize more efficient constraint checking in detecting context inconsistencies.

A. *Motivation and Pilot Study*

We conducted a pilot study to investigate how severe the link redundancy problem is in constraint checking. The study was conducted in an exhaustive way, in the sense that it enumerated all possible consistency constraints composed of various formula types. To control the study cost, we restricted the constraint height (as in the syntax tree or CCT) to no more than four. This left us a total of 1,658 constraints. To conduct the truth value evaluation and link generation, we allowed $bfunc$ terminals to return random values. This simulated all conditions about how contexts participated into $bfunc$ calculations. Then we measured and found that all existing constraint checking techniques (all based on Complete Link Generation or CG, as aforementioned) suffered from the link redundancy problem: 88% constraints with over 50% redundant links, and 73% constraints with over 75% redundancy. This strongly calls for research efforts to reduce or eliminate such unnecessary overhead in the constraint checking.

There is only one piece of work addressing this problem. We named it Optimized Link Generation) (or OG) [23], following its own name of optimized constraint checking (OCC). However, its goal of being *optimized* is far from the reality, as OG relies only on static formula type information in the constraints to analyze and prune potential redundant link generation. To play safe, the analysis is conservative. For example, its paper [23] reported that the use rate of its generated links is only 83.3% for "and"/"or"/"implies" formulas (or 16.7% redundancy rate). Besides, it only made a formula-type based prediction for the link use rate, and one does not know its practical effectiveness. When tested with our preceding example constraint in Fig. 4, OG has a redundancy rate as high as still 53.3% (i.e., not effective for this particular example). When tested with the 1,658 constraints in our pilot study, OG still caused a severe, although alleviated, redundancy link problem: 80% constraints (88% for CG) with over 50% redundant links, and 57% constraints (73% for CG) with over 75% redundancy. Therefore, this further calls for more effective link redundancy techniques.

In this paper, we would next propose a novel technique, named Minimized Link Generation (or MG, as aforemen-



Consistency constraints
MG-deployed constraint checking
Derivation of Substantial Conditions (§III.C)
Minimized Link Generation
Truth Value Evaluation
Tainting CCT for S-CCT (§III.D)
S-CCT-guided Link Gen (§III.E)
Detected context inconsistencies
Contexts (changing)
Complete Link Generation (CG)
original constraint checking

Fig. 6: MG overview

tioned), to minimize the link redundancy in the constraint checking. MG exploits a key data structure, named substantial CCT (or S-CCT for short), which isolates a CCT into two parts, one concerning substantial nodes that contribute to the calculation of final links, while the other concerning immaterial ones that are irrelevant to final links. For the constraint in our illustrative example in Fig. 4, S-CCT marks only nodes 1–7 in the CCT, and thus MG generates links associated with these nodes only. The S-CCT can also automatically adjust itself based on runtime information about contexts under checking, adaptive to various constraints and their evaluations. In theory, MG can identify and eliminate the generation of all redundant links, as we discuss later.

B. Technique Overview

We now give an overview on how the MG technique works in Fig. 6. It consists of three steps. First, MG derives *substantial conditions* with respect to different formula types used in given consistency constraints. There conditions characterize how a formula type contributes to this formula's violation or satisfaction. Second, with derived substantial conditions, MG taints the CCTs associated with the constraints in a top-down manner to obtain sub trees, named S-CCTs, which contain only those nodes that generate intermediate links necessary for the calculation of final links. Finally, MG conducts the link generation for the nodes on the S-CCTs only, without affecting the final results. MG promises to generate more effective links than CG (as illustrated by the dashed arrows and process).

In a CCT as in Fig. 4, nodes are at different layers. We use notation $e_1 \succ e_2$ to denote that node e_2 is node e_1's direct child, and use notation $\succ_l$ and $\succ_r$ to distinguish the left and right children when necessary. As our later analysis relies only on the formula type f and truth value tv associated with each node, we can simplify a node as $e = (f, tv)$ when no ambiguity. For example, nodes 1 and 2 in Fig. 4 can be simplified as: $\textsf{node1} = (\forall, \text{False})$, and $\textsf{node2} = (\text{implies}, \text{False})$, with relations $\textsf{node1} \succ \textsf{node2}$ or $\textsf{node1} \succ_l \textsf{node2}$.

Our key is to identify the nodes on a CCT that will contribute to the calculation of final links on the root node, i.e., identifying those nodes on the corresponding S-CCT. We realize this by first deriving substantial conditions for the formula type associated with each node, as explained below.

C. Derivation of Substantial Conditions

According to its associated formula type (i.e., "∀", "∃", "and", "or", "implies", or "not"), a node can contribute differently to its parent node's violation or satisfaction in a CCT.

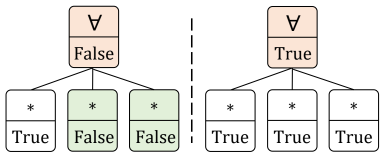


Fig. 7: Two typical cases of substantial nodes for the ∀ formula

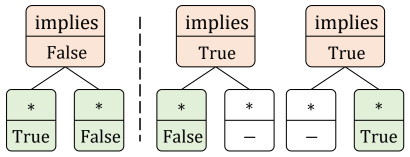


Fig. 8: Three typical cases of substantial nodes for the implies formula

Take the universal (∀) formula as an example, as illustrated in Fig. 7, which gives two typical cases. If the root node has a truth value of False (i.e., formula violated), and only its child node(s) being False contribute(s) to its violation. In this case, we consider the child node(s) being False as *substantial node(s)*, and model its(their) condition(s) as *substantial condition(s)* in a form of pair $\ulcorner$*selection*, *requirement*$\urcorner$, with *selection* to select the target child nodes and *requirement* to set up a condition to qualify such nodes. For this case, the root node's substantial condition is $\ulcorner \succ, \text{False} \urcorner$, meaning: (1) selecting all its child nodes, and (2) qualify them by truth value False. We derive this substantial condition by $(\forall, \text{False}) \xrightarrow{SC} \ulcorner \succ, \text{False} \urcorner$, as listed in Fig. 9. Then for the second case in Fig. 7, the root node has a truth value of True (i.e., formula satisfied), and all its child nodes (being True) together contribute to its satisfaction, but no single one fully decides it. Thus, we remove these child nodes from the consideration of being substantial, with the corresponding substantial condition as $(\forall, \text{False}) \xrightarrow{SC} \emptyset$, suggesting no further analysis.

The substantial conditions for the existential (∃) formula can be derived similarly: $(\exists, \text{True}) \xrightarrow{SC} \ulcorner \succ, \text{True} \urcorner$, and $(\exists, \text{False}) \xrightarrow{SC} \emptyset$, corresponding to its two similar cases.

We illustrate the implies formula in Fig. 8. If the root node has a truth value of False (i.e., formula violated), its left child node being True and right child node being False together contribute to this violation. Thus, both child nodes

$$(\forall,\ \text{True}) \xrightarrow{SC} \emptyset,\ (\forall,\ \text{False}) \xrightarrow{SC} \ulcorner\succ, \text{False}\urcorner;$$
$$(\exists,\ \text{True}) \xrightarrow{SC} \ulcorner\succ, \text{True}\urcorner,\ (\exists,\ \text{False}) \xrightarrow{SC} \emptyset;$$
$$(\text{and},\ \text{True}) \xrightarrow{SC} \ulcorner\succ, *\urcorner,\ (\text{and},\ \text{False}) \xrightarrow{SC} \ulcorner\succ, \text{False}\urcorner;$$
$$(\text{or},\ \text{True}) \xrightarrow{SC} \ulcorner\succ, \text{True}\urcorner,\ (\text{or},\ \text{False}) \xrightarrow{SC} \ulcorner\succ, *\urcorner;$$
$$(\text{implies},\ \text{True}) \xrightarrow{SC} \ulcorner\succ_l, \text{False}\urcorner,\ \ulcorner\succ_r, \text{True}\urcorner,$$
$$(\text{implies},\ \text{False}) \xrightarrow{SC} \ulcorner\succ, *\urcorner;$$
$$(\text{not},\ *) \xrightarrow{SC} \ulcorner\succ, *\urcorner.$$

Fig. 9: Substantial condition derivation

are substantial. Otherwise, the root node is True, and either its left child node is False or right child node is True, any one of which would decide the root node's truth value. Thus, the corresponding child node(s) satisfying such conditions is(are) substantial. Altogether, we derive the substantial conditions as: $(\text{implies, False}) \xrightarrow{SC} \ulcorner\succ, *\urcorner$ ("$*$" is the wildcard character), and $(\text{implies, True}) \xrightarrow{SC} \ulcorner\succ_l, \text{False}\urcorner, \ulcorner\succ_r, \text{True}\urcorner$.

The substantial conditions for the "and", and "or" formulas can be similarly derived and are thus omitted for discussion. Finally, for the not formula, since its node contains one child node only, which must be substantial, we derive its substantial condition as: $(\text{not}, *) \xrightarrow{SC} \ulcorner\succ, *\urcorner$.

We list all derived substantial conditions in Fig. 9.

D. Tainting CCT for S-CCT

With the derived substantial conditions for all pairs of formula type and truth value in Fig. 9, we now proceed to introduce how to taint a CCT to obtain its corresponding S-CCT, in order to guide later link generation without redundancy. We name this process *conditional tainting*, which works after the truth value evaluation (i.e., truth values available), but before the link generation in constraint checking.

The conditional tainting aims to taint all the nodes in a CCT that satisfy substantial conditions. Specially, for any node a in the CCT (starting from the root node, in a top-down manner), we examine all its child nodes to see whether any of them satisfies a's substantial condition. If yes, this child node is tainted. Otherwise, all child nodes are not tainted and we stop further examining these nodes. Therefore, all eventually tainted nodes must be connected, which are known as the S-CCT (a sub-tree of the original CCT).

This tainting process works by a DFS algorithm, as in Algorithm 1. It (GETSCCT) starts with the root node of the given CCT, by feeding the node to the tainting logic (TAINT) when the whole constraint is violated (i.e., the root node's truth value being False). Then, for any fed node, the TAINT procedure checks this node's all child nodes to see whether any of them satisfies the fed node's associated substantial condition. If yes, the concerned child node is tainted and added into the target S-CCT (cct_s). For the CCT in Fig. 4, the tainting process starts from node 1 $(\forall, \text{False})$, whose substantial condition is $\ulcorner\succ, \text{False}\urcorner$. Therefore, only those child nodes with a truth value of False are tainted, i.e., node 2 $(\text{implies}, \text{False})$. Then, the process further examines the child nodes of the newly tainted node 2, whose substantial condition

Algorithm 1 Conditional Tainting

```
procedure GETSCCT(cct)
    if ISROOTVIOLATED(cct.root) then
        return TAINT(cct.root)
    end if
end procedure
procedure TAINT(currentNode)
    $cct_s \leftarrow \{currentNode\}$
    for $c \in currentNode.children$ do
        if SATISFY(currentNode, c) then
            $subResult \leftarrow$ TAINT(c)
            $cct_s \leftarrow cct_s \cup subResult$
        end if
    end for
    return $cct_s$
end procedure
```

is $\ulcorner\succ, *\urcorner$, and taints nodes 3 and 5 accordingly. Similarly, nodes 4, 6, and 7 are also tainted, and then this tainting process terminates. The finally obtained S-CCT contains a total of seven nodes (i.e., nodes 1–7), as earlier illustrated by the shadowed sub-tree in Fig. 4.

E. S-CCT-guided Link Generation

Following the nodes tainted in the S-CCT, the link generation can now be conducted with the guidance of avoiding generating redundant links. This process is straightforward. For any node in a CCT, if the node is tainted (i.e., in the S-CCT), links are still generated using the original semantics in Fig. 5. Otherwise, the node is not tainted, and no link is generated for this node. This process would guide to generate links only for nodes in the S-CCT (e.g., node 1–7 as in Fig. 4). Note that the S-CCT would be updated when its corresponding CCT evolves, and this update is efficient as shown in Algorithm 1.

To understand how our MG differs from existing work on the link generation (e.g., CG and OG), we illustrate their capabilities in Fig. 10.. In the figure, *all-links* refer to the links generated by CG as in Fig. 5, which can be divided into two parts, namely, *must-links* and *may-links*. Must-links refer to the links that have to be generated so as to calculate the final links at the root node of a CCT, while may-links refer to the remaining links, the avoidance of whose generation will not affect final links (i.e., redundant links, as aforementioned). In the comparison, CG generates all-links, including all must-links and may-links; OG reduces part of may-links by its static redundancy analysis, but still leaves some redundancy, i.e., including all must-links and some may-links. Our MG

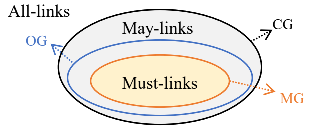


Fig. 10: Relations among different types of links (all-links = must-links + may-links)

instead eliminates all may-links, thus including only all must-links. This is guaranteed by the following two theorems, namely, *soundness* (for correctness) and *completeness* (for minimization) theorems.

Theorem 1 (**Soundness**). *MG generates all must-links.*

Theorem 2 (**Completeness**). *MG generates no may-links.*

Sketch of proof: We prove Theorems 1 and 2 together. For saving space, we give the sketch of proof by induction. The intuition is to show that for any node that generates links necessary for calculating the final links, MG would taint it, and also taint those of its child nodes as long as they generate links necessary for calculating this node's links.

Base step. Consider the root node. According to Algorithm 1, MG taints it only when the constraint is violated. In this case, all the root node's generated links are considered as the final links. Therefore, they are naturally must-links.

Inductive step. Consider a given node m, whose generated links are must-links. We now examine its child nodes. We discuss only the $\forall$ and $\mathrm{implies}$ nodes, and other nodes are similar.

1) When m is a $\forall$ node, according to the substantial conditions in Fig. 9, if this node's truth value is False, MG would taint only those of its child nodes being False, i.e., $(\forall, \mathrm{False}) \xrightarrow{SC} \ulcorner \succ, \mathrm{False} \urcorner$. This essentially corresponds to the link generation semantics for the $\forall$ formula in Fig. 5, i.e., $\mathcal{L}\left[\forall v \in \mathrm{C}(f)\right]_{\alpha} = \{\langle \mathsf{vio}, \{v := \mathrm{e}\}\rangle \otimes \mathcal{L}[f]_{\alpha[v:=\mathrm{e}]} \mid \mathrm{e} \in \mathrm{C} \wedge \mathcal{T}[f]_{\alpha[v:=\mathrm{e}]} = \mathrm{F}\}$. Therefore, MG taints right those child nodes that would contribute necessary links (i.e., must-links).
2) When m is an $\mathrm{implies}$ node, we look into its link generation semantics in Fig. 5. There are four cases. We observe that if m's truth value is False, its left child must be True and right child be False (case (1)). Then all its child nodes' links (i.e., $\mathcal{L}\left[f_1\right]_{\alpha}$ and $\mathcal{L}\left[f_2\right]_{\alpha}$) are necessary. For this case, MG right taints both its child nodes, as the substantial conditions in Fig. 9. For the remaining three cases, m's truth value is True, and m's links depend on its left child node only if the node is False (cases (2) and (4)), and on its right child node only if the node is True (cases (2) and (3)) in Fig. 5. This again exactly corresponds to MG's substantial conditions in Fig. 9. Therefore, MG also taints m's child nodes generating must-links.

With other types of nodes similarly proved, we combine both the base and inductive steps, and have that MG generates must-links only (i.e., no may-link). This completes the proof. □

F. Application to Constraint Checking

MG minimizes the links that have to be generated in the constraint checking, and is generic in that it can be easily applied to existing constraint checking techniques, e.g., ECC (entire checking) [8], PCC (partial checking) [8], and Con-C (concurrent checking) [18]. We use a combination to refer to such applications, e.g., ECC-CG (ECC with CG applied, i.e., original ECC) and ConC-MG (Con-C with MG applied, i.e., MG replacing the original CG).

Applications to ECC and Con-C. These applications are straightforward. ECC and Con-C use our aforementioned CG semantics in Fig. 5. Therefore, MG can be directly applied by using the substantial condition derivation (in Fig. 9) and conditional tainting (in Algorithm 1) to decide the S-CCT and then using the S-CCT to generate links, to replace the original CG.

Application to PCC. This application needs a little discussion. PCC differs from ECC and Con-C, in that the former sometimes reuses its previously calculated links, while the latter always regenerate them (easy for the replacement from CG to MG). In each scheduled constraint checking, PCC does not destroy its maintained CCTs, but updates part of them. Therefore, when applying MG to PCC, one needs to consider two cases: (1) when generating links for the CCT part that is updated by PCC in this round, MG generates links only for tainted nodes; (2) when reusing previous links on the CCT part that is not updated by PCC in this round, MG checks whether these links are ready (possibly not ready if the concerned nodes are not tainted in the last round): if ready, then reuse these links; otherwise, generate links for tainted nodes. In other words, MG still generates must-links, but additionally takes care of the reuse and late generation issue for PCC.

IV. Evaluation

In this section, we evaluate and compare our MG to existing work on the effectiveness of link generation, as well as the benefits to constraint checking.

A. Research Questions

We aim to answer the following three research questions:

RQ1 (Motivation) How does existing link generation (CG and OG) in constraint checking suffer from the link redundancy problem?

RQ2 (Effectiveness) How effective is MG in reducing redundant link generation, as compared to CG and OG?

RQ3 (Benefits) How does MG's link generation contribute to the efficiency improvement of existing constraint checking (with ECC, PCC, and Con-C)?

We answer RQ1–RQ2 by a comparative study with exhaustively synthesized constraints and controlled factors, and answer RQ3 by a case study with real-world data. We explain the evaluation design and setup below.

B. Evaluation Design and Setup

1) RQ1 and RQ2: We study the characteristics of existing link generation in constraint checking on the link redundancy. First, we select the consistency constraints for study. We choose to exhaustively enumerate all possible constraints by a guided synthesis. The synthesis explores all possible formula types and combinations from the constraint language by a controlled limit, i.e., by the height of a constraints abstract tree. The synthesis guarantees that each formula type has been tried at each possible place in a constraint, and this gave use a total of 1,658 well-formed constraints, when limiting the height to no more than four[1]. On one hand, it is already a sufficient number of various constraints for analysis, covering all kinds of formula types and combinations. On the other hand, if one extends the height limit to 5, that would drastically add over 10^8 constraints, which overwhelms any possible analysis.

Second, we decide how to calculate *bfunc* values in the constraints. As all constraints are synthesized, *bfunc* terminals do not carry real semantics. Still, one can simulate their value calculations by: (1) enumerating the elements in contexts used in the constraints, say, from 2 to 20 (controlled in feeding context changes to the constraints), and (2) randomly returning truth values for *bfunc* calculations by a controlled probability p, say, from 0.01 to 0.99 (keeping consistent for the same context values as parameters).

In the study, we compare three link generation techniques, namely, CG, OG, and MG, control the number limit l of elements in contexts, from 2, 5, 10, 15, to 20, and control the probability p for *bfunc* to return True at 0.01, 0.05, 0.1, 0.3, 0.5, 0.7, 0.9, 0.95, to 0.99. These factors are designed as independent variables.

For each configuration decided by these independent variables, we repeat experiments 5,000 times to alleviate possible bias caused by randomness. With averaged calculations over the 5,000 runs for each configuration, we calculate the metric of *link utilization rate* (or ULR) to measure the proportion of the links actually used by calculating final links against all generated links by a specific technique.

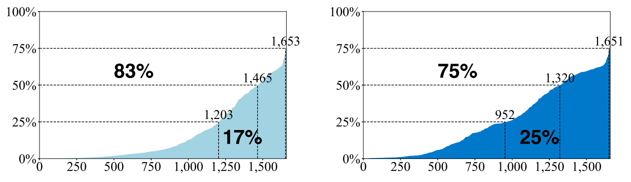
To answer RQ1, we compare CG and OG to show how severe they suffer from the link redundancy problem and study their characteristics. To answer RQ2, we compare our MG to CG and OG to show its effectiveness in identifying and eliminating redundant link generation.

2) RQ3: We study how MG contributes to the efficiency improvement for constraint checking. We followed existing work [8], [18]–[20] to use a real-world application SmartCity with its large-volume taxi-driving data for the case study. The application was accompanied with 22 consistency constraints, covering all seven formula types, and a total of 1.55 million taxi data, covering 760 vehicles within a continuous period of 24 hours. The taxi data were transformed into the form of 6.75 million context changes fed to constraint checking for detecting context inconsistencies.

We conduct constraint checking by combining three link generation techniques (CG, OG, and our MG) with existing constraint checking techniques (ECC, PCC, and Con-C). The latter has different levels of efficiency, and we thus study how MG can additionally boost their efficiency on the base of CG and OG. We measure the efficiency improvement by time cost, and storage improvement by memory cost.

All experiments were conducted on a commodity PC with one Intel Core i7-10750H CPU @ 2.60GHz and 15GiB RAM. The PC was installed with Ubuntu 21.10 and Java SE 17.0.1.

[1]These resources are released at https://github.com/cychen2021/issre22.



(a) CG (b) OG

Fig. 11: Averaged ULRs for CG and OG

C. Result Analyses

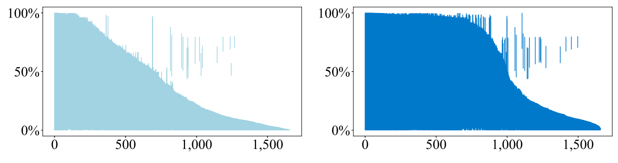
1) RQ1 (Motivation): Fig. 11 illustrates the averaged ULR measures for CG and OG across all 1,658 consistency constraints. We order these constraints according to their ascending ULR tends for CG and OG, respectively, for better illustration. We also mark the 25%, 50%, 75% quantiles by dashed lines for reference.

From Fig. 11, we observe that both CG and OG suffered seriously from the link redundancy problem, resulting in very low averaged ULR measures, e.g., [<1%, 77%] for CG and [<1%, 91%] for OG. In particular, 1,203 (72.6%) and 952 (57.4%) constraints have averaged ULR measures less than 25% (i.e., link redundancy over 75%), and only 5 (0.3%) and 7 (0.4%) constraints have averaged ULR measures over 75% (i.e., link redundancy less than 25%), for CG and OG, respectively. When accumulating the area above and below each curve, CG caused a total of 83% redundant links and OG still caused 75% redundant links for all constraints. Therefore, when taking into account all types of constraints, CG seriously suffers from the link redundancy problem, and OG improves a little but the benefits are very limited. This strongly calls for the new efforts to identify and eliminate such redundancy and such efforts must be flexible to cope with all types of constraints.

We also illustrate the ULR ranges for CG and OG across these 1,658 constraints in Fig. 12, which were caused under different settings (e.g., different contexts and $bfunc$ results). From the figure, we observe that both CG and OG are very unstable in generating links in terms of link redundancy. For example, around half of all constraints have a ULR range over 50% for both CG and OG. This suggests that even for a single constraint, a good link generation technique has to be flexible to cope with its dynamic information (e.g., contexts and $bfunc$ results), so as to realize an overall high ULR. This also echoes our MG's idea that combines both static (constraint type and syntax) and dynamic (runtime truth values) analysis in identifying and removing redundant link generation.

Therefore, we conclude that both CG and OG suffer from severe link redundancy problem and new research efforts must take care of static and dynamic analysis in the constraint checking to achieve the identification and elimination of redundant link generation.

2) RQ2 (Effectiveness): Fig. 13 compares the averaged ULR measures for CG, OG, and MG across all 1,658 constraints. This gives an intuitive and exhaustive picture of



(a) CG (b) OG

Fig. 12: ULR ranges for CG and OG

how a specific link generation technique works for all types of constraints. We make the comparisons aligned for each constraint to better illustrate the differences among the three techniques (Green: CG, Orange: OG, and Blue: MG).

Regarding the averaged ULR measures, we observe that CG ranges from <1% to 77% and OG ranges from <1% to 91%. Although the overall improvement is clear, the ULR gaps between CG and OG are very inconsistent with respect to different types of constraints. This suggests that different constraints imposed different challenges for reducing redundant link generations, and a sole static analysis technique like OG cannot cope with all situations. On the other hand, for our MG technique, it achieved a landslide victory by reaching an always 100% ULR measure, as it promised. This suggests that MG realizes both successfully identifying all redundant link generations and automatically adapting to different constraints according to their inherent characteristics. We owe the ability to MGs dedicatedly designed static-dynamic hybrid analysis. Note that MG's absolute improvements on the averaged ULR measures can be 23–100% (mean: 83%) over CG and 9–100% (mean: 75%) over OG, which are impressive.

Therefore, we conclude that MG can identify and eliminate all link redundancy in the constraint checking and are capable of adapting to all constraint types.

3) RQ3 (Benefits): Fig. 14 compares the time costs in the link generation for CG, OG, and MG under the real-world application scenario with 22 consistency constraints and 6.75 million context data.

From the figure, we observe that when combined with different constraint checking techniques, although CG, OG, and MG incurred different time costs, MG always worked most efficiently. For example, MG spent only 0.14–0.34 minutes, while CG spent 2.41–70.75 minutes and OG spent 2.11–67.53 minutes. We note that all the three techniques generated exactly the same final links in the constraint checking (all correct), and thus MG's efficiency improvements on the link generation totally attributes to its greatly reduced link redundancy.

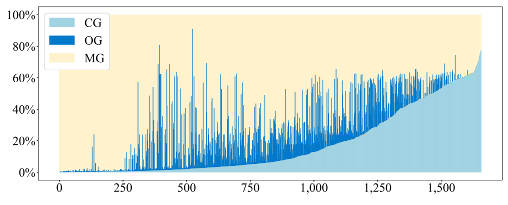


Fig. 13: ULR comparison for CG, OG, and MG

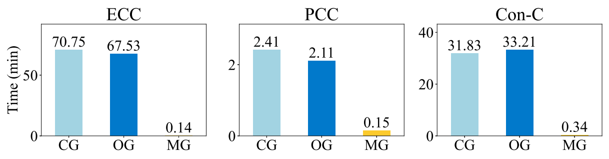


Fig. 14: Time cost comparison for CG, OG, and MG

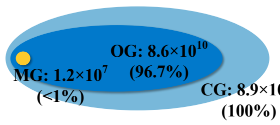


Fig. 15: Illustration of generated links for CG, OG, and MG

In particular, MG reduced 93.8–99.8% time cost (or 15–504x speedup) over CG, and 92.9–99.8% time cost (or 13–481x speedup) over OG, respectively. We owe MG's significant time reduction on the link generation to its dramatically removed redundant link generation. To see it, we illustrate CG's, OG's, and MG's generated links in Fig. 15. We observe that MG's links (ULR = 100%) occupy only <1% of CG's links (ULR = 0.13%), while OG's links (ULR = 0.14%) occupy 96.7% of CG's links. MG's differences from CG and OG are indeed huge. Besides, MG's time reduction over existing work (CG and OG) is comparable (with similar orders of magnitude) to its MG's link reduction, and this suggests that MG's internal S-CCT maintenance overhead is extremely small.

As aforementioned, the link generation is only part of the whole constraint checking, which also includes the truth value evaluation. Therefore, we also studied how MG's improvement on the link generation helps towards the improvement on the checking efficiency of the whole constraint checking. Note that the truth value evaluation is not affected by MG, and thus MG's contribution could be alleviated. Still, we observe from the measurement that MG reduced 26.2–45.2% time cost over CG for the whole constraint checking, and 22.3–45.4% time cost over OG, respectively. Note that this achievement was obtained over MG's internal overhead, which is extremely small, almost negligible (second-level). This also suggests that as a kernel step in the constraint checking, the improvement on the link generation can indeed bring additional benefits to existing constraint checking techniques, and the benefits can apply to all such techniques in a generic and transparent way.

Therefore, we conclude that MG can bring significant efficiency improvements on the link generation (15–504x over CG and 13–481x over OG), and promising improvements even on the whole constraint checking (26.2–45.2% time reduction over CG and 22.3–45.4% time reduction over OG).

D. Threat Analyses

One threat concerns the external validity of our experiments and conclusions, since we used synthesized consistency constraints. To alleviate its impact, on one hand, we exhaustively synthesized all well-formed constraints for analysis. Although the height limit is only four, the constraints' number reaches 1,658, and their types and combinations are complete for analysis. Besides, we used synthesized constraints only in the comparative study. We, on the other hand, also validated MG's effectiveness and compared to existing techniques under a real-world application scenario with actual constraints and millions of context data. The validation obtained consistent results, echoing our earlier analysis in the comparative study. Finally, to avoid possible bias, we re-implemented all link generation and constraint checking techniques under the same I/O interface and data structures. We would later release our implementations for follow-up research.

V. Related Work

Our work relates to existing research efforts on the consistency management of software artifacts, which can be subject to slow evolution or frequent changing.

Managing the consistency for such software artifacts effectively support the applications that run with them for adaptive or smart services. Traditional software artifacts are typically static or evolving slowly, which include XML documents [6], [26], [27], UML models [28]–[30], set-and-relation-based models [31], workflows [32], and distributed algorithms [33]. Managing the consistency for such artifacts focuses mainly on the reliability. Some new software artifacts that need the consistency management require to take care of a new requirement, efficiency, as such artifacts are typically dynamic or even changing frequently. One example of such artifacts is application contexts, which guide how an application reacts to environmental changes and include many application scenarios, e.g., Humanoid Companion Robot [34], Pollen Wise [35], and self-driving vehicle systems [1], [2]. Some middleware infrastructures have even be developed to dedicatedly support such context-aware computing, e.g., Cabot [24], Adam [36], Lime [37], and CARISMA [38].

Managing the consistency for application contexts needs to identify problems in the contexts collected or derived from environmental sensing or application executions. One way is to deploy data-centric approaches to identify anomalies in context data, i.e., cleaning noise in raw sensory data (e.g., missing and cross reads in RFID) [39]–[43], by filtering [39], fuzzy matching [44], sequence-based rules [40], watermark generation [45], or probabilistic methods [46].

The other way is to detect inconsistencies in the contexts from the perspective of consistency constraints [6], [13], [47], [48], which specify necessary properties about application scenarios. This way is more generic, and has been extensively studied. Its recent focus is on the efficiency, so that applications can realize problems in its contexts and resolve them in a timely manner [15]–[17]. Researchers have proposed various acceleration techniques for speeding up the constraint checking for efficient context inconsistency detection, e.g., full checking (ECC) [6], incremental checking (PCC) [8], CPU-based parallel checking (Con-C) [18], and GPU-based parallel checking (GAIN) [19]. One interesting line of work supporting this aspect is to selectively decide the timepoints the constraint checking should be scheduled, so that unnecessary scheduling can be suppressed [9], [20], thus contributing to more efficient inconsistency detection in an indirect way.

Our work in this paper opens a new direction to support more efficiency context inconsistency detection by removing redundant link generation during constraint checking. Some redundancy-reduction work from other fields has also been studied, echoing our efforts in this work, e.g., avoiding redundant table scans to speed up the SQL-MapReduce task translation [49], avoiding redundant computations to speed up GNN training [50], and simplifying floating-point computations to speed up look-table operations in neural networks [51]. Although the principles are similar, these efforts are closely bound to their subjects, not applicable to our problem. In this paper, we propose a substantial condition derivation based technique MG to particularly address the link redundancy problem in constraint checking. As reported in the experiments, our work MG can achieve tens or hundreds of times of speedup for link generation, and generally apply to existing constraint checking techniques to bring extra efficiency benefits by up to 45.4% time reduction. Therefore, this direction would be promising to be combined with other improvement aspects to together contribute to even more efficiency of context inconsistency detection, thus guarding the reliability of adaptive applications running in dynamic environments.

VI. Conclusion

In this paper, we tackled the link redundancy problem in constraint checking, and proposed a novel technique, MG, to automatically identify and remove redundant link generation, without harming any checking result. We theoretically proved MG's soundness and completeness, and conducted both a comparative study with synthesized consistency constraints and a case study with large-volume real-world context data. The study results validated MG's effectiveness in eliminating all link redundancy and improving the efficiency by 15–504x on link generation over existing work. MG also supported context inconsistency detection by an additional efficiency gain up to 45.4% time reduction, automatically applicable to all existing constraint checking techniques.

We are seeking ways to further validate MG under more complex constraints (e.g., height over four) and other application scenarios (e.g., unmanned drone and self-driving vehicle context data). We also explore ways to automatically reformulate constraints to be redundancy-free by construction, rather than runtime remedy, which could be even more cost-effective for applications.

Acknowledgment

This research was supported by the Natural Science Foundation of Jiangsu Province under Grant Nos. BK20202001 and BK20220771, and the Natural Science Foundation of China under Grant Nos. 61932021 and 62072225. The authors would also like to thank the support from the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

References

[1] "Califronia DMV. 2020. autonomous vehicle disengagement reports," 2020. [Online]. Available: https://www.dmv.ca.gov/portal/file/2020-autonomous-vehicle-disengagement-reports-csv/

[2] D. Shepardson, H. Jin, and J. White, "Self-driving car companies zoom ahead, leaving U.S. regulators behind," *Reuters*, Feb. 2022. [Online]. Available: https://www.reuters.com/business/autos-transportation/self-driving-car-companies-zoom-ahead-leaving-us-regulators-behind-2022-02-02/

[3] C. Ke, F. Xiao, Z. Huang, and F. Xiao, "A user requirements-oriented privacy policy self-adaption scheme in cloud computing," *Frontiers of Computer Science*, vol. 17, no. 2, p. 172203, 2023.

[4] A. Harter, A. Hopper, P. Steggles, A. Ward, and P. Webster, "The anatomy of a context-aware application," *Wireless Networks*, vol. 8, no. 2, pp. 187–197, 2002, publisher: Springer.

[5] A. H. Van Bunningen, L. Feng, and P. M. Apers, "Context for ubiquitous data management," in *International Workshop on Ubiquitous Data Management*. IEEE, 2005, pp. 17–24.

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

[7] C. Xu, S.-C. Cheung, and W.-K. Chan, "Incremental consistency checking for pervasive context," in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 292–301.

[8] C. Xu, S.-C. Cheung, W.-K. Chan, and C. Ye, "Partial constraint checking for context consistency in pervasive computing," *ACM Transactions on Software Engineering and Methodology*, vol. 19, no. 3, pp. 1–61, 2010.

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

[10] Z. Mao, Y. Gu, B. Jiang, D. Xu, X. Sun, and W. Liu, "Incipient fault diagnosis for high-speed train traction systems via improved lstm," *Scientia Sinica Informationis*, vol. 51, no. 6, pp. 997–1012, May 2021, original document in Chinese.

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

[12] I. Park, D. Lee, and S. J. Hyun, "A dynamic context-conflict management scheme for group-aware ubiquitous computing environments," in *29th Annual International Computer Software and Applications Conference (COMPSAC'05)*, vol. 1. IEEE, 2005, pp. 359–364.

[13] Y. Bu, T. Gu, X. Tao, J. Li, S. Chen, and J. Lu, "Managing quality of context in pervasive computing," in *2006 Sixth International Conference on Quality Software (QSIC'06)*. IEEE, 2006, pp. 193–200.

[14] J. Chomicki, J. Lobo, and S. Naqvi, "Conflict resolution using logic programming," *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 1, pp. 244–249, 2003.

[15] C. Xu, X. Ma, C. Cao, and J. Lu, "Minimizing the side effect of context inconsistency resolution for ubiquitous computing," in *International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*. Springer, 2011, pp. 285–297.

[16] C. Xu, S.-C. Cheung, W.-K. Chan, and C. Ye, "On impact-oriented automatic resolution of pervasive context inconsistency," in *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering: companion papers*, 2007, pp. 569–572.

[17] ——, "Heuristics-based strategies for resolving context inconsistencies in pervasive computing applications," in *2008 The 28th International Conference on Distributed Computing Systems*. IEEE, 2008, pp. 713–721.

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

[19] J. Sui, C. Xu, W. Xi, Y. Jiang, C. Cao, X. Ma, and J. Lu, "GAIN: GPU-based constraint checking for context consistency," in *Proceedings of the 21st Asia-Pacific Software Engineering Conference*, vol. 1, Jeju, South Korea, Dec. 2014, pp. 319–326.

[20] H. Wang, C. Xu, B. Guo, X. Ma, and J. Lu, "Generic adaptive scheduling for efficient context inconsistency detection," *IEEE Transactions on Software Engineering*, vol. 47, no. 03, pp. 464–497, mar 2021.

[21] L. Zhang, H. Wang, C. Xu, and P. Yu, "INFUSE: Towards efficient context consistency by incremental-concurrent check fusion," in *Proceedings of the 38th International Conference on Software Maintenance and Evolution (ICSME 2022)*, Limassol, Cyprus, Oct. 2022, forthcoming.

[22] C. Xu, Y. Qin, P. Yu, C. Cao, and J. Lu, "Techniques for growing software: Paradigm and beyond," *Scientia Sinica Informationis*, vol. 50, no. 11, pp. 1595–1611, Nov. 2020, original document in Chinese.

[23] C. Xu, S.-C. Cheung, and W.-K. Chan, "Goal-directed context validation for adaptive ubiquitous systems," in *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*. Minneapolis, Minnesota, USA: IEEE Computer Society, 2007, pp. 1–10.

[24] C. Xu, S.-C. Cheung, C. Lo, K.-C. Leung, and J. Wei, "Cabot: On the ontology for the middleware support of context-aware pervasive applications," in *IFIP International Conference on Network and Parallel Computing*. Springer, 2004, pp. 568–575.

[25] J. Sui, C. Xu, S.-C. Cheung, W. Xi, Y. Jiang, C. Cao, X. Ma, and J. Lu, "Hybrid cpu–gpu constraint checking: Towards efficient context consistency," *Information and Software Technology*, vol. 74, pp. 230–242, 2016.

[26] S. P. Reiss, "Incremental maintenance of software artifacts," *IEEE Transactions on Software Engineering*, vol. 32, no. 9, pp. 682–697, 2006.

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

[28] A. Egyed, "Instant consistency checking for the UML," in *Proceedings of the 28th International Conference on Software Engineering*, 2006, pp. 381–390.

[29] X. Blanc, I. Mounier, A. Mougenot, and T. Mens, "Detecting model inconsistency through operation-based model construction," in *Proceedings of the 30th ACM/IEEE International Conference on Software Engineering*. IEEE, 2008, pp. 511–520.

[30] "ArgoUML." [Online]. Available: https://github.com/argouml-tigris-org/argouml

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

[32] C. Chen, C. Ye, and H.-A. Jacobsen, "Hybrid context inconsistency resolution for context-aware services," in *2011 IEEE International Conference on Pervasive Computing and Communications (PerCom)*. IEEE, 2011, pp. 10–19.

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

[34] P.-H. Kuo, S.-T. Lin, J. Hu, and C.-J. Huang, "Multi-sensor context-aware based chatbot model: An application of humanoid companion robot," *Sensors*, vol. 21, no. 15, p. 5132, 2021.

[35] "Pollen Wise - What's in your air, when and where." [Online]. Available: https://play.google.com/store/apps/details?id=com.PollenSense.PollenWise

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

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

[38] L. Capra, W. Emmerich, and C. Mascolo, "CARISMA: Context-aware reflective middleware system for mobile applications," *IEEE Transactions on software engineering*, vol. 29, no. 10, pp. 929–945, 2003.

[39] S. R. Jeffery, M. Garofalakis, and M. J. Franklin, "Adaptive cleaning for RFID data streams," in *Vldb*, vol. 6. Citeseer, 2006, pp. 163–174.

[40] 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*. Citeseer, 2006, pp. 175–186.

[41] K. T. Patil, V. Bansal, V. Dhateria, and S. K. Narayankhedkar, "Probable causes of RFID tag read unreliability in supermarkets and proposed solutions," in *2015 International Conference on Information Processing (ICIP)*. IEEE, 2015, pp. 392–397.

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

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

[44] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani, “Robust and efficient fuzzy match for online data cleaning,” in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 2003, pp. 313–324.

[45] Y. Song, Y. Li, H. Yang, J. Xu, Z. Luan, and W. Li, “Adaptive watermark generation mechanism based on time series prediction for stream processing,” *Frontiers of Computer Science*, vol. 15, no. 6, p. 156213, 2021.

[46] N. Khoussainova, M. Balazinska, and D. Suciu, “Towards correcting input data errors probabilistically using integrity constraints,” in *Proceedings of the 5th ACM international workshop on Data engineering for wireless and mobile access*, 2006, pp. 43–50.

[47] Y. Bu, S. Chen, J. Li, X. Tao, and J. Lu, “Context consistency management using ontology based model,” in *International conference on extending database technology*. Springer, 2006, pp. 741–755.

[48] C. Xu and S.-C. Cheung, “Inconsistency detection and resolution for context-aware middleware support,” *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 336–345, 2005.

[49] R. Lee, T. Luo, Y. Huai, F. Wang, Y. He, and X. Zhang, “YSmart: Yet another SQL-to-MapReduce translator,” in *2011 31st International Conference on Distributed Computing Systems*, Jun. 2011, pp. 25–36.

[50] Z. Jia, S. Lin, R. Ying, J. You, J. Leskovec, and A. Aiken, “Redundancy-free computation for graph neural networks,” in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD’20)*. New York, NY, USA: Association for Computing Machinery, Aug. 2020, pp. 997–1005.

[51] M. S. Razlighi, M. Imani, F. Koushanfar, and T. Rosing, “LookNN: Neural network with no multiplication,” in *Design, Automation and Test in Europe Conference and Exhibition, 2007*, Mar. 2017, pp. 1775–1780.