Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

RESEARCH ARTICLE

MG+: Towards Efficient Context Inconsistency Detection by Minimized Link Generation

Chuyang Chen[1,2] | Huiyan Wang[1,2] | Lingyu Zhang[1,2] | Chang Xu[1,2] | Ping Yu[1,2]

[1]State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China

[2]School of Computer Science, Nanjing University, Nanjing, Jiangsu, China

Correspondence
Huiyan Wang and Chang Xu, Nanjing University, Nanjing, Jiangsu, China.
Email: why@nju.edu.cn and changxu@nju.edu.cn

Abstract

Self-adaptive applications are becoming increasingly attractive, with the ability to smartly understand their runtime environments (or contexts) and deliver adaptive services, e.g., location-aware navigation or resource-sensitive suggestions. However, due to inherent noises in the process of sensing and interpreting environmental information, there is a growing demand for guarding the consistency of collected contexts to avoid application misbehavior and, at the same time, minimize extra costs. Existing work attempted to achieve this by speeding up the kernel constraint checking module inside the consistency guarding process. Most of these efforts were spent on reusing previous checking results or parallelizing the checking process, but they all leave one central step of constraint checking, i.e., link generation, untouched. In this step, the checking engine provides reasons to explain the violation of constraints under check. It occupies a substantial part of the total time cost. Focusing on this key link generation step, we proposed MG, which deploys a rigorous analysis to automatically identify and avoid redundancy in the link generation without harming any correctness of the checking results. MG has been proven sound (always guaranteeing correctness) and complete (entirely removing redundancy). Moreover, based on our observation that MG's redundancy elimination also assists another core step of constraint checking to reduce unnecessary computation further, we additionally enhance MG with an escape-condition optimization to escape unnecessary evaluation of truth values to further improve the efficiency of constraint checking in an aspect other than link generation. We call it MG+ for distinguishing. Our experiments with synthesized and real-world consistency constraints reported that, compared with existing work, MG eliminates all link redundancy (83% to 0), and based on it, MG+ further reduces significant truth value calculations (e.g., 49.74% reduction when combined with ECC and Con-C). Generally, MG brought 14x–500x speed-ups in link generation, and MG+ further made 1.2x–1.9x speed-ups in truth value evaluation. Altogether, MG reduced the total constraint checking time up to 45.4%, and MG+ reduced it up to 61.0%.

KEYWORDS:
context inconsistency, constraint checking, link redundancy

1 | INTRODUCTION

The rapid advancement of modern sensing and actuating technologies has led to a surge in the popularity of self-adaptive applications across various domains, including location-aware navigators, self-driving vehicles [1,2], cloud computing systems [3,4], and mobile apps [5,6,7]. These applications offer intelligent services by analyzing and interpreting their operational environments, referred to as *contexts* [8], and adjusting their behaviors accordingly. However, the presence of unavoidable and uncontrollable environmental noise can cause discrepancies in the application contexts from their actual states, potentially resulting in incorrect behaviors (for example, inaccurate location reporting [9,10,11,12,13,14]) or system failures.

Contexts naturally lack oracles to assess their correctness. Thus, various studies explored ways to detect and trace flaws in contexts. One promising approach is checking the contexts against pre-specified rules, namely, *consistency constraints* [9,10,11], that should hold under laws in the application domain and the physical world—e.g., changes of a vehicle's location should never be faster than light. We should consider any violation of such constraints as a *context inconsistency* [9,10,11] and report it to later resolution [15,16,17,18,19,20,21,22]. As the constraint checking process aims to guard an application's reliability by timely detecting and handling anomalies at runtime, it should be *effective* and *efficient* to avoid compromising the application's normal functionalities [10,11,23,24,25,26]. Researchers proposed various techniques to this end, varying from simple correctness baseline (Entire Constraint Checking, ECC [9]), to delicate incremental computation (Partial Constraint Checking, PCC [11]) and CPU/GPU-based parallelization (Con-C [23] and GAIN [24]).

It is extremely difficult to further improve the performance of constraint checking upon all these edge technologies. However, the growing complexity and dynamicity of contexts has exceeded the capability of existing techniques [27]. In some scenarios with ubiquitous cyber-physical interactions and huge-volume data, these techniques may miss over 90% inconsistencies [25]. The performance of constraint checking techniques is confronted with a staggeringly huge challenge.

In this article, we approach the issue with a novel perspective. We categorize existing research endeavors, such as incremental or parallel constraint checking, under the "making-it-faster" category. This means they focus on accelerating the constraint checking process by speeding up every step in the process. Conversely, we introduce a new category, "making-it-less," where our aim is to decrease the computational workload by identifying and removing redundant computations that do not impact checking results. If successful, this innovative approach could complement existing methods in a unique, orthogonal manner, elevating their performance to unprecedented heights.

We dig into the constraint checking process and observe a two-step pattern that comprises all existing constraint checking techniques, namely, *truth value evaluation* and *link generation*. The former step examines whether a given consistency constraint is violated respecting the contexts under check by a truth value of True or False. The latter step generates a data structure named *link* [9,10,11] to explain why the violation happens by identifying elements that contribute to the truth value. With the links, users are able to locate and correct errors in the contexts. The link generation step consumes a large proportion (up to 45% according to our later experiments) of the total checking time cost. However, a large part of links generated as intermediate results in this step are redundant (23–100% according to the experiments) in the sense that they never affect the checking results (detailed analysis in Section 3). Our "making-it-less" conjecture hinges on this observation. If possible, tracing and avoiding such redundancy would lead to substantial efficiency improvements. Based on such an insight, the article studies the redundant link problem and proposes a sound (always checking correctly) and complete (removing all redundancy) technique to prevent them.

Our proposed Minimized Link Generation technique (or MG) automatically identifies redundant links and prevents them from being generated in advance. Compared to the existing practice of link generation, e.g., Complete Link Generation (or CG) used in existing constraint checking techniques [9,11,23], MG significantly reduces link generation by 23–100%. It achieves this via a hybrid static-dynamic analysis, which first constructs a data structure named S-CCT that encodes a constraint's static syntax information and then evolves it with dynamic truth value information associated with the contexts during checking. We prove that the S-CCT can mark all necessary generation sites for the final checking results, i.e., with these sites, we can guarantee the correctness of the final results (*soundness*). We also prove that MG is *complete*, i.e., able to identify *all* redundant links and prevent them from being generated. Moreover, MG is *generic* by design and handily applicable to all existing constraint checking techniques. Furthermore, we observe a surprising benefit brought by MG: S-CCTs can also assist another core step, truth value evaluation, of constraint checking. It indicates possibly unused truth values that do not get involved in computing the truth value of the final result and are not required by any link (because these links are eliminated by the S-CCTs). Based on this observation, we derive escape conditions to mark some of such truth values and avoid the evaluation of them. This optimization further

boosts the efficiency of constraint checking. We add this extension to MG and devise an even more powerful technique MG+. With these analyses and optimization, MG and MG+ significantly improve the performance of constraint checking techniques.

Our evaluation showed that: (1) MG realizes 100% link utilization (i.e., removing 100% redundant link generation), as compared to existing work, which encounters severe link redundancy problem where 75–83% links are redundant; (2) When applied to existing constraint checking techniques (e.g., ECC[9], PCC[11], and Con-C[23]), MG brings tens to hundreds times speed-ups in link generation, and reduces the total constraint checking time up to 45.4%; (3) The evaluation also exhibits clear benefits brought by MG+ to existing constraint checking techniques, which further reduces the constraint checking time up to 61.0%.

In summary, we make the following contributions in this article, where the second one and part of the third one are extended over our previous work[28]:

- We propose Minimized Link Generation (MG) for efficient context inconsistency detection that can identify and avoid redundant link generation in constraint checking and theoretically prove its soundness and completeness.
- We extend MG to MG+ with an escape-condition optimization for further efficiency in the truth value evaluation step and also theoretically prove its correctness.
- We evaluate the performance of MG and MG+ and validate their effectiveness in minimizing link generation and reducing truth value evaluation. We confirm that they lead to substantial efficiency improvements over existing constraint checking techniques.

The remainder of this article is organized as follows. Section 2 uses an illustrative example to introduce the problem and technical background. Section 3 reports a pilot study to motivate our work and then elaborates on our MG technique to identify and eliminate redundant link generation. Section 4 extends MG to MG+, which can further reduce truth value evaluation by an escape-condition optimization. Section 5 evaluates MG and MG+ under controlled experiments with exhaustive constraint analysis and a case study with real-world data. Section 6 discusses the related work in recent years, and finally, Section 7 concludes this article.

2 | BACKGROUND

In this section, we introduce preliminary concepts of constraint checking by an illustrative example.

2.1 | Preliminary

Context and context pool. A *context* embodies information about an application's runtime environment[25]. It can be modeled as a finite set containing multiple elements. Each of these elements reflects a facet of the whole context. Consider a highway charging system that tracks information about a traveling vehicle, including its license plate number, driving speed, and location, and calculates highway tolls accordingly. The context that models vehicles recently driving by a highway gantry a is $\mathrm{C}_{\mathsf{a}} = \{\mathsf{car}_1, \mathsf{car}_2, \cdots\}$, wherein each element car_i identifies one vehicle with some specific license plate number, driving speed, and other information. As time elapses, contexts adapt to the changing environment, e.g., some cars may leave the highway, and some new cars enter. Contexts relevant to the application are collected in a *context pool*. Respecting the highway charging system, contexts associated with all the highway gantries constitute the system's context pool.

Consistency constraint. For the highway charging system, gantries deploy cameras and sensors to track vehicles, but the tracking process may be subject to sensor disturbance. The contexts are thus inaccurate and/or incomplete and may even conflict with each other, leading to *context inconsistency*[9,10,11]. To address context inconsistency, we formulate *consistency constraints* to detect errors in the contexts. An example constraint $\mathrm{R}_{\mathrm{exit}}$ is as follows:

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

As illustrated in Fig. 1, the constraint requires a vehicle at Gantry "out" to enter the highway either from Gantry "rampA" or "rampB," but not both. Any counter-example (the red car in Fig. 1) is impossible and should be considered as a sensor error. Otherwise, the system may charge a false toll.

We use first-order logical formulas to specify such consistency constraints, following previous work[9,10,11,23,29,25,30,31]. Therein, C is a context, v_i is a variable that takes an element from C as its value, and the terminal $bfunc$ is an application-specific predicate that returns a Boolean value (True/T or False/F):

$$f ::= \forall v \in \mathrm{C}(f) \mid \exists v \in \mathrm{C}(f) \mid (f) \text{ and } (f) \mid$$
$$(f) \text{ or } (f) \mid (f) \text{ implies } (f) \mid \text{not } (f) \mid$$
$$bfunc\left(v_1, v_2, \cdots, v_n\right) \mid \mathrm{T} \mid \mathrm{F}.$$

We can also use a syntax tree to represent a consistency constraint, e.g., Fig. 2 for $\mathrm{R}_{\mathrm{exit}}$.

Constraint checking. *Constraint checking* examines the contexts given in a context pool against consistency constraints to report *truth values* (indicating whether any context inconsistencies happen) and *links* (indicating how context inconsistencies happen, if any). Two kernel steps of constraint checking, *truth value evaluation* and *link generation*, compute these results respectively. We introduce them below.

2.2 | Constraint Checking

Consider the previous consistency constraint $\mathrm{R}_{\mathrm{exit}}$. Let $C_{\mathrm{out}} = \{\mathrm{car}_1, \mathrm{car}_2\}$, $C_{\mathrm{rampA}} = \{\mathrm{car}_1\}$, and $C_{\mathrm{rampB}} = \{\mathrm{car}_1, \mathrm{car}_2\}$. These contexts return a False truth value for $\mathrm{R}_{\mathrm{exit}}$, indicating that a context inconsistency happens. They also give a link of $\langle \text{violated}, \{v_1 = \mathrm{car}_1, v_2 = \mathrm{car}_1, v_3 = \mathrm{car}_1\}\rangle$, which means that the element car_1 in contexts C_{out}, C_{rampA}, and C_{rampB} causes the violation (and other elements are innocent).

For ease of presentation, let $f_A = \exists v_2 \in \mathrm{C}_{\mathrm{rampA}}\left(\mathrm{sameCar}\left(v_1, v_2\right)\right)$ and $f_B = \exists v_3 \in \mathrm{C}_{\mathrm{rampB}}\left(\mathrm{sameCar}\left(v_1, v_3\right)\right)$, and the previous constraint becomes $\mathrm{R}_{\mathrm{exit}} = \forall v_1 \in \mathrm{C}_{\mathrm{out}}\left(\left(f_A\right) \text{ implies } \left(\text{not } \left(f_B\right)\right)\right)$.

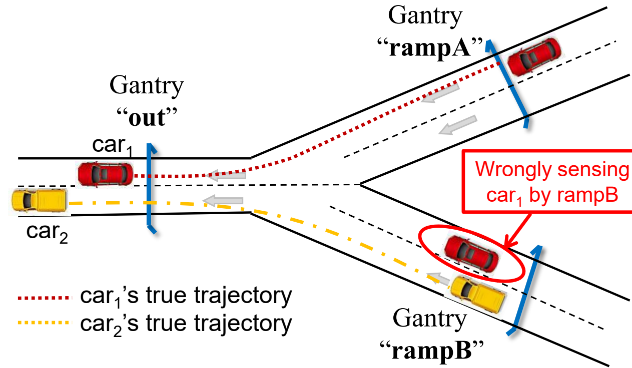


Figure 1 Illustration of a highway scenario

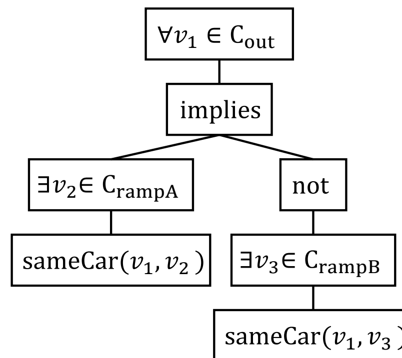


Figure 2 Syntax tree structure of constraint $\mathrm{R}_{\mathrm{exit}}$

2.2.1 | Truth Value Evaluation

The truth value of a constraint is evaluated according to the semantics listed in Fig. 3.$\mathcal{T}[f]_\alpha$ denotes the truth value of formula f under the variable assignment α. The semantics leverages a recursive definition.

To evaluate R_{exit}, we first enumerate all possible assignments to v_1 in the top universal formula to instantiate its subformula (the implies formula):

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

Take the situation where $v_1 := \text{car}_1$ as an example:

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

f_A and f_B are similarly evaluated according to the semantics in Fig. 3:

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

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

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

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

We explain the above calculation in a top-down manner for human readability. However, in the actual system, it is conducted in a bottom-up manner by a post-order traversal: a node is evaluated after that all its children have been evaluated. Expanding the constraint's syntax tree respecting all possible variable assignments provides a better illustration for the process, as shown in Fig. 4. The expanded tree is named *consistency computation tree* (CCT) as in previous work[10,11,23,29,25,24,31]. Truth values for each node are annotated beside them in the figure. During constraint checking, a post-order traversal 4→3→7→8→6→5→2→... evaluates these truth values in sequence until the final result (False at the root node numbered 1).

2.2.2 | Link Generation

Given these truth values, we can generate links to explain why a formula is violated (when its node has an False truth value) or satisfied (when its node has a True truth value).

Link generation also follows a post-order traversal on CCT according to the semantics in Fig. 5. For the example in Fig. 4, all links generated during the process are annotated beside each node.

In Fig. 5, $\mathcal{L}[f]_\alpha$ denotes the generated links for formula f under the variable assignment α, and there are some auxiliary functions and operators. FlipSet is used to flip the violated/satisifed tag for all links in a given set (from violated to satisfied, or vice versa), and $\otimes$ is the Cartesian product used to merge two link set. For example, the Cartesian product for two link sets S_1

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

Figure 3 Truth value evaluation semantics

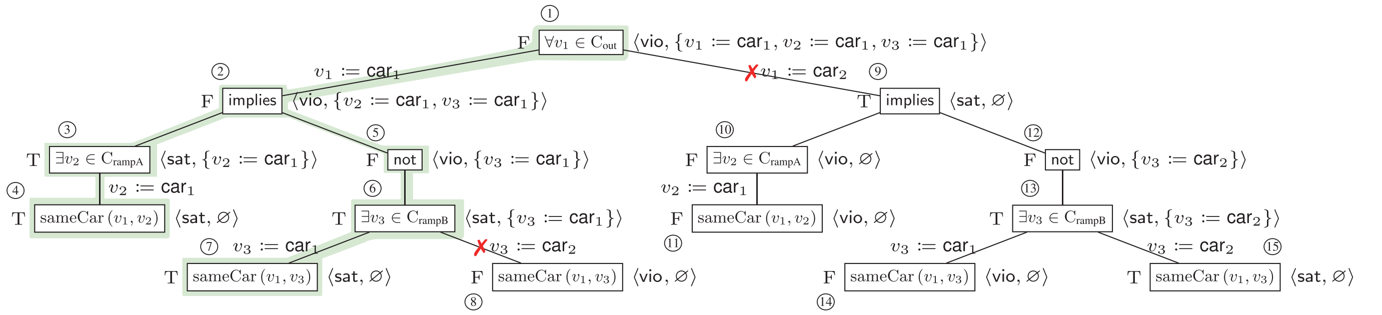


Figure 4 CCT for the example constraint with S-CCT shadowed

and S_2 are:

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

where $l_1 \cdot l_2 = \langle Tag(l_1), Bindings(l_1) \cup Bindings(l_2) \rangle$, with $Tag(l)$ referring to l's violated/satisfied tag, and $Bindings(l)$ referring to l's assignment pairs. Note that we use $\{\langle \text{vio/sat}, \varnothing \rangle\}$ to denote a link set with only a violated/satisfied tag and no assignment pair (no variable bound to any specific value). It is treated as $\varnothing$ when computing unions and Cartesian products.

The final result of the link generation process is the link $\langle \text{violated}, \{v_1 = \text{car}_1, v_2 = \text{car}_1, v_3 = \text{car}_1\} \rangle$ at the root node numbered 1. The link's interpretation contains two parts: 1) the violated tag indicates that the constraint is violated, i.e., a context inconsistency happens, and 2) the assignment pairs $\{v_1 = \text{car}_1, v_2 = \text{car}_1, v_3 = \text{car}_1\}$ means that assigning car_1 to v_1, v_2, v_3 causes this inconsistency.

We observe that to calculate the final link at Node 1 in Fig. 4, all nodes of the tree participated in the computation and generated their links. Comparing the semantics in Fig. 3 and Fig. 5, link generation is much more complicated than truth value evaluation. However, many nodes generate unnecessary links. For the example in Fig. 3.1, the final link at Node 1 only depends on links at Node 2–7, and links at Node 8–15 do not affect it at all. The redundancy rate (53.3%) is rather high!

3 | MINIMIZED LINK GENERATION

In this section, we report a pilot study to motivate our work and elaborate on our MG technique that eliminates redundancy in link generation to boost the efficiency of constraint checking.

3.1 | Motivation and Pilot Study

We conduct a pilot study to investigate the severity of the redundancy problem. We conduct the study in an exhaustive manner by enumerating all possible consistency constraints constructed from different types of formulas. We limit the heights of these

$$\mathcal{L}\,[\forall v \in \mathrm{C}(f)]_\alpha = \{\langle \textsf{violated}, \{v := \mathrm{e}\}\rangle\} \otimes \left\{\mathcal{L}[f]_{\alpha[v:=\mathrm{e}]} \middle| \mathrm{e} \in \mathrm{C} \wedge \mathcal{T}[f]_{\alpha[v:=\mathrm{e}]} = \mathrm{F}\right\}$$
$$\mathcal{L}\,[\exists v \in \mathrm{C}(f)]_\alpha = \{\langle \textsf{satisfied}, \{v := \mathrm{e}\}\rangle\} \otimes \left\{\mathcal{L}[f]_{\alpha[v:=\mathrm{e}]} \middle| \mathrm{e} \in \mathrm{C} \wedge \mathcal{T}[f]_{\alpha[v:=\mathrm{e}]} = \mathrm{T}\right\}$$
$$\mathcal{L}\left[(f_1)\ \textsf{implies}\ (f_2)\right]_\alpha =$$
$$\begin{aligned}
&(1)\ \textsf{FlipSet}\left(\mathcal{L}\left[f_1\right]_\alpha\right) \otimes \mathcal{L}\left[f_2\right]_\alpha,\ \text{if}\ \mathcal{T}\left[f_1\right]_\alpha = \mathrm{T}\ \text{and}\ \mathcal{T}\left[f_2\right]_\alpha = \mathrm{F}\\
&(2)\ \textsf{FlipSet}\left(\mathcal{L}\left[f_1\right]_\alpha\right) \cup \mathcal{L}\left[f_2\right]_\alpha,\ \text{if}\ \mathcal{T}\left[f_1\right]_\alpha = \mathrm{F}\ \text{and}\ \mathcal{T}\left[f_2\right]_\alpha = \mathrm{T}\\
&(3)\ \mathcal{L}\left[f_2\right]_\alpha,\ \text{if}\ \mathcal{T}\left[f_1\right]_\alpha = \mathcal{T}\left[f_2\right]_\alpha = \mathrm{T}\\
&(4)\ \textsf{FlipSet}\left(\mathcal{L}\left[f_1\right]_\alpha\right),\ \text{if}\ \mathcal{T}\left[f_1\right]_\alpha = \mathcal{T}\left[f_2\right]_\alpha = \mathrm{F}
\end{aligned}$$
$$\mathcal{L}\left[(f_1)\ \textsf{and}\ (f_2)\right]_\alpha =$$
$$\begin{aligned}
&(1)\ \mathcal{L}\left[f_1\right]_\alpha \otimes \mathcal{L}\left[f_2\right]_\alpha,\ \text{if}\ \mathcal{T}\left[f_1\right]_\alpha = \mathcal{T}\left[f_2\right]_\alpha = \mathrm{T}\\
&(2)\ \mathcal{L}\left[f_1\right]_\alpha \cup \mathcal{L}\left[f_2\right]_\alpha,\ \text{if}\ \mathcal{T}\left[f_1\right]_\alpha = \mathcal{T}\left[f_2\right]_\alpha = \mathrm{F}\\
&(3)\ \mathcal{L}\left[f_2\right]_\alpha,\ \text{if}\ \mathcal{T}\left[f_1\right]_\alpha = \mathrm{T}\ \text{and}\ \mathcal{T}\left[f_2\right]_\alpha = \mathrm{F}\\
&(4)\ \mathcal{L}\left[f_1\right]_\alpha,\ \text{if}\ \mathcal{T}\left[f_1\right]_\alpha = \mathrm{T}\ \text{and}\ \mathcal{F}\left[f_2\right]_\alpha = \mathrm{T}
\end{aligned}$$
$$\mathcal{L}\left[(f_1)\ \textsf{or}\ (f_2)\right]_\alpha =$$
$$\begin{aligned}
&(1)\ \mathcal{L}\left[f_1\right]_\alpha \otimes \mathcal{L}\left[f_2\right]_\alpha,\ \text{if}\ \mathcal{T}\left[f_1\right]_\alpha = \mathcal{T}\left[f_2\right]_\alpha = \mathrm{F}\\
&(2)\ \mathcal{L}\left[f_1\right]_\alpha \cup \mathcal{L}\left[f_2\right]_\alpha,\ \text{if}\ \mathcal{T}\left[f_1\right]_\alpha = \mathcal{T}\left[f_2\right]_\alpha = \mathrm{T}\\
&(3)\ \mathcal{L}\left[f_1\right]_\alpha,\ \text{if}\ \mathcal{T}\left[f_1\right]_\alpha = \mathrm{T}\ \text{and}\ \mathcal{T}\left[f_2\right]_\alpha = \mathrm{F}\\
&(4)\ \mathcal{L}\left[f_2\right]_\alpha,\ \text{if}\ \mathcal{T}\left[f_1\right]_\alpha = \mathrm{T}\ \text{and}\ \mathcal{F}\left[f_2\right]_\alpha = \mathrm{T}
\end{aligned}$$
$$\mathcal{L}\,[\textsf{not}\ (f)]_\alpha = \textsf{FlipSet}\left(\mathcal{L}[f]_\alpha\right)$$
$$\mathcal{L}\left[bfunc(v_1, \dots, v_n)\right]_\alpha =$$
$$\begin{aligned}
&(1)\ \{\langle \textsf{satisfied}, \varnothing\rangle\}, \text{if}\ \mathcal{T}\left[bfunc(v_1, \dots, v_n)\right]_\alpha = \mathrm{T}\\
&(2)\ \{\langle \textsf{violated}, \varnothing\rangle\}, \text{if}\ \mathcal{T}\left[bfunc(v_1, \dots, v_n)\right]_\alpha = \mathrm{F}
\end{aligned}$$

Figure 5 Link generation semantics

constraints to no more than 4 to ensure an acceptable computation cost. A total of 1,658 constraints are left after this filtering process. During checking, $bfunc$ terminals return random truth values to simulate overall situations respecting the value of a $bfunc$ in different contexts. By measurement, we observe that all existing constraint checking techniques with complete link generation (aforementioned CG) suffer from severe link redundancy problems: 88% constraints waste over 50% links, and 73% constraints waste over 75% links. Such terrible statistics strongly call for a new technique to reduce or eliminate these unnecessarily generated links.

Only one piece of previous work centers its effort on this problem. For consistency, we rename it Optimized Link Generation (OG), following the original name Optimized Constraint Checking (OCC)[29]. However, its "optimization" is far from enough. The core step of OG lies in a static analysis that conservatively considers redundancy that may happen on different types of formulas. Such consideration deviates a lot from the reality. First, the article itself reports an average link redundancy rate of 16.7% respecting and/or/implies formulas based on theoretical deduction[29]. Second, in actual constraint checking scenarios, the leftovers that cannot be optimized by OG are even worse. For example, OG is absolutely ineffective for the constraint in Fig. 4 (reducing no redundancy). When being applied to the 1,658 constraints in our pilot study, OG slightly improves the performance, but the remaining redundancy is still severe: 80% constraints waste over 50% links (the number is 88% for CG), and 57% constraints waste over 75% links (the number is 73% for CG). These data impose urges for a more effective link generation technique.

This article will propose a novel technique, Minimized Link Generation (MG), to resolve this problem. MG exploits a key data structure, the Substantial CCT (S-CCT), that combines static and dynamic information to identify links that will be redundant and prevent their generation. An S-CCT evolves according to static deduction rules and dynamic runtime truth values and can separate a CCT into relevant and irrelevant parts, respecting the final result. By restricting link generation to S-CCTs, we can generate actually necessary links while eliminating all redundancy. When being applied to the illustrative example in Fig. 4,

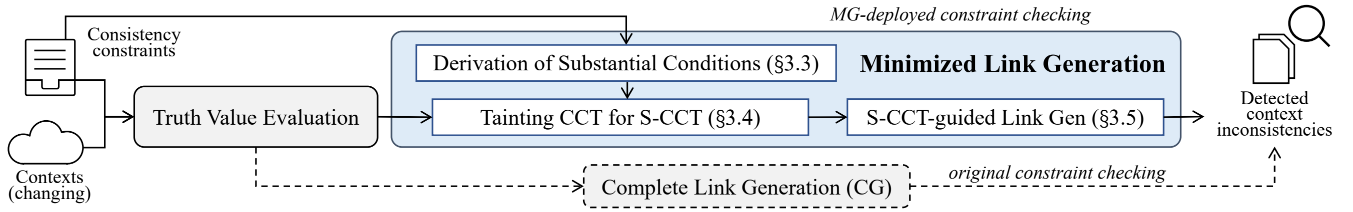


Figure 6 MG overview

MG constructs an S-CCT with Node 1–7 marked, and they are exactly where necessary links forms. Left Node 8–15 are not on the S-CCT, and as previously analyzed, links generated on these nodes make no contribution to the final result.

The hybrid static-dynamic S-CCT analysis offers adaptability for MG. It can attain optimal performance under various situations. We will prove this in Section 3.5. Moreover, S-CCTs bring us an unexpected benefit. Other than the link generation step, which S-CCTs are designed for, the truth value evaluation can also make use of the information. We will extend our MG to MG+ to further improve the overall efficiency of constraint checking in Section 4.

3.2 | Methodology Overview

We give an overview of how MG works in Fig. 6. The part in the blue box is our MG technique, and the part with dashed lines is the original CG technique that is replaced by MG. It consists of three steps. First, MG derives *substantial conditions* respecting different types of formula. These conditions characterize how a specific type of formula will contribute to the final links during constraint checking. Second, MG taints a CCT to get an S-CCT. An S-CCT is the subtree of the tainted CCT, which contains all nodes that will affect the final result during link generation, i.e., with and only with links generated on these S-CCT nodes, we can compute the correct final link. Finally, MG conducts link generation restricted onto the S-CCT. Due to the aforementioned S-CCT properties, the process is guaranteed to generate the correct result without any redundancy.

CCT nodes are at different layers. We use $e_1 \succ e_2$ to denote that a node e_2 is a direct child of a node e_1, $\succ_l$ and $\succ_r$ to differentiate left and right children if necessary. Sometimes, it is sufficient for analysis with only the formula type and the truth value information of a CCT node. If so, we simplify the representation of the node to (f, tv) where f is the formula type and tv is the truth value. For example, Node 1 and Node 2 in Fig. 4 can be simply represented by: $\texttt{node1} = (\forall, \text{False})$ and $\texttt{node2} = (\text{implies}, \text{False})$, and the relations between them are $\texttt{node1} \succ \texttt{node2}$ or $\texttt{node1} \succ_l \texttt{node2}$.

The key of our MG technique is identifying CCT nodes that will actually affect the final result during link generation. These nodes form the S-CCT. To realize this, we must first derive conditions under which a CCT node should be an S-CCT node. These conditions are called substantial conditions, and their derivation is elucidated in the next section.

3.3 | Derivation of Substantial Conditions

Nodes with different formula types (viz., "∀," "∃," "and" "or," "implies," and "not") contribute differently to the constraint checking results of their parents and the whole CCT.

Take universal (∀) formulas as an example. Fig. 7 shows different cases when checking a universal formula. When the universal formula has a truth value of False (i.e., violation happens), only child nodes with also a False truth value contribute to the violation. In this case, child nodes with a False truth value are *substantial nodes*, and vice versa; substantial nodes must meet this condition of producing a False truth value. The condition is what we call a *substantial condition*, written as $(\forall, \text{False}) \xrightarrow{SC}$

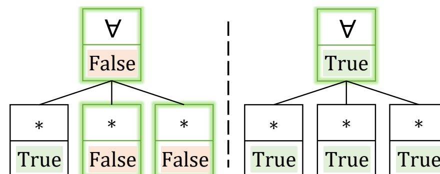


Figure 7 Two typical cases of substantial nodes for the ∀ formula

$\ulcorner \succ, \text{False} \urcorner$. It means that when a node is a universal node with a False truth value (the left-hand side), we select its child nodes with also False truth values as the substantial nodes (the right-hand side).

Another case happens when the universal node has a True truth value (i.e., the formula is satisfied). In this case, all its child nodes (which must be with True truth values) together contribute to the result, but no single one can fully decide it. Thus, we avoid further exploring these nodes, and the corresponding substantial condition is $(\forall, \text{False}) \xrightarrow{SC} \emptyset$, meaning that no further analysis on the children.

Derivation of the substantial conditions of existential formulas are similar. They are $(\exists, \text{True}) \xrightarrow{SC} \ulcorner \succ, \text{True} \urcorner$ and $(\exists, \text{False}) \xrightarrow{SC} \emptyset$.

Another illustrative example is the "implies" formulas, as shown in Fig. 8. When an "implies" node is with a False truth value, its left child must be with a True truth value, and its right child must be with a False truth value. They together decide the truth value of the "implies" node—changing either of them will cause the truth value of the "implies" node an alternation. In this case, both child nodes are substantial. Otherwise, the root node is with a True truth value. Either the left child is with a False truth value, or the right child is with a True truth value. As long as either of the two condition holds, the truth value of the "implies" node stays unchanged, however the truth value of another child alters. Thus, we derive two substantial conditions $(\text{implies, False}) \xrightarrow{SC} \ulcorner \succ, * \urcorner$ and $(\text{implies, True}) \xrightarrow{SC} \ulcorner \succ_l, \text{False} \urcorner, \ulcorner \succ_r, \text{True} \urcorner$ ("$*$" is a wildcard matching any cases).

Derivation of substantial conditions for the "and" and "or" formulas is similar.

Finally, a "not" node has only one child, and its truth value is totally determined by that child, so the child is always a substantial node, and the corresponding substantial condition is $(\text{not}, *) \xrightarrow{SC} \ulcorner \succ, * \urcorner$.

Fig. 9 lists all derived substantial conditions.

3.4 | Tainting CCTs for S-CCTs

We proceed to taint CCTs using substantial conditions derived in Section 3.3. After tainting, we will obtain the S-CCT for a CCT, i.e., the necessary part of the CCT that actually contributes to the final result of link generation. It can be used to guide the link generation step to eliminate unnecessary computation (detailed in the next section). We name this process *conditional tainting*. It is inserted between the truth value evaluation step and the link generation step, so truth values needed by the analysis are available, and the obtained S-CCT can later guide link generation.

Conditional tainting aims to taint all nodes of an S-CCT. For any node of the CCT (starting from the root node, in a top-down manner), we examine its children against substantial conditions. We recursively taint children that satisfy any substantial

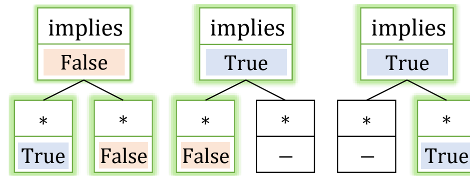


Figure 8 Three typical cases of substantial nodes for the implies formula

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

Figure 9 Deriving substantial conditions

condition and skip the analysis on other children. Besides, if a node is skipped, all its descendants will be skipped, too. Tainted nodes will eventually form a sub-tree of the original CCT, namely, the S-CCT.

Algorithm 1 Conditional Tainting

```
 1: procedure GETSCCT(cct)
 2:     if ISROOTVIOLATED(cct.root) then
 3:         return TAINT(cct.root)
 4:     end if
 5: end procedure
 6: procedure TAINT(currentNode)
 7:     cct_s ← {currentNode}
 8:     for c ∈ currentNode.children do
 9:         if SATISFY(currentNode, c) then
10:             subResult ← TAINT(c)
11:             cct_s ← cct_s ∪ subResult
12:         end if
13:     end for
14:     return cct_s
15: end procedure
```

Alg. 1 conducts the tainting process by depth-first search. It (GETSCCT) starts from the root node if a violation happens (Line 2–4) and then advances by feeding each node to the tainting logic (Line 10). The tainting logic (TAINT) checks whether any child of a node satisfies one of the substantial conditions and further explores that child if so. Otherwise, the child and its descendants are skipped. We visualize the algorithm on the CCT in Fig. 4. It starts from Node 1 ($\forall$, False) whose substantial condition is $\ulcorner >, \text{False}\urcorner$. Only children with a False truth value (Node 2) meet the condition. After tainting Node 2, the algorithm further examines its children, Node 3 and Node 5. The substantial condition of Node 2 is $\ulcorner >, *\urcorner$. Both Node 3 and Node 5 meet it. Similarly, Node 4, 6, and 7 are tainted, and then the algorithm terminates. Eventually, we get an S-CCT of seven nodes (Node 1–7), illustrated by the shadowed sub-tree in Fig. 4.

3.5 | S-CCT-guided Link Generation

We can now use the S-CCT to guide our link generation to avoid link redundancy. This process is straightforward. For any node on a CCT, if it is tainted (i.e., on the S-CCT), we generate links as usual, using the semantics listed in Fig. 5. Otherwise, we simply skip it. We theoretically guarantee (detailed later) that nothing abnormal will happen for the final result. The final links will keep their correctness as before. In summary, the S-CCT-guided link generation can be regarded as if we restrict the original link generation process from the whole CCT onto the S-CCT.

The S-CCT would be updated as the CCT evolves, and the updating is as efficient as Alg. 1.

Fig. 10 gives an illustration of how MG differs from existing link generation techniques (CG and OG). In the figure, *all-links* refer to the links generated by CG (all links in Fig. 4). They can be divided into two parts: *must-links* and *may-links*. Must-links refer to the links that must be generated insofar as we want to compute a correct final result, and may-links refer to the remaining

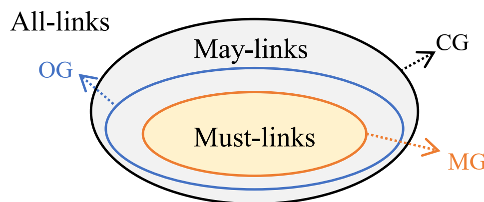


Figure 10 Relations among different types of links (all-links = must-links + may-links)

links. The absence of may-links will not have any effect on the final result, i.e., they are redundant. In the comparison, CG generates all-links, including all must-links and may-links; OG reduces part of may-links by its static analysis but still leaves much redundancy, i.e., it generates all must-links and many (though not all) may-links. MG instead eliminates all may-links and generates only must-links. Such property of our MG technique is guaranteed by two following theorems, viz., the soundness theorem (for correctness) and the completeness theorem (for minimization).

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 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 Theorem 1 and Theorem 2 together. To save space, we give a sketch of the proof, which is briefly a proof by induction. The intuition is to show that MG will taint and only taint nodes that generate necessary links for the final result. For any such node, MG will similarly taint and only taint its children, which generate necessary links for the middle result on that node. The two propositions form an inductive relation that applies to all tainted nodes to guarantee that they will only generate necessary links for the final result.

Base step. Consider the root node. According to Alg. 1, MG taints it only if a violation happens. In this case, links generated by the root node must be taken as the final result. They are naturally must-links.

Inductive step. Consider a given node *m* that generates must links. We examine its children. We only discuss the cases where *m* is a $\forall$ or "implies" node. Other cases are similar.

1. When *m* is a $\forall$ node, according to the substantial conditions in Fig. 9, if *m*'s truth value is False, MG will only taint children with a False truth value according to $(\forall, \text{False}) \xrightarrow{SC} \ulcorner >, \text{False} \urcorner$. This essentially corresponds to the link generation semantics for $\forall$ formulas in Fig. 5, $\mathcal{L}\,[\forall v \in \mathrm{C}(f)]_\alpha = \{\langle \mathsf{vio}, \{v := \mathrm{e}\}\rangle\} \otimes \mathcal{L}[f]_{\alpha[v:=\mathrm{e}]} | \mathrm{e} \in \mathrm{C} \wedge \mathcal{T}[f]_{\alpha[v:=\mathrm{e}]} = \mathrm{F}$. MG taints right those children that generate must-links

2. When *m* is an "implies" node, we delve 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 $\mathtt{True}$, and the right child must be False (Case 1). Links of both children (i.e., $\mathcal{L}\left[f_1\right]_\alpha$ and $\mathcal{L}\left[f_2\right]_\alpha$) are necessary. In this case, MG correctly taints these two nodes (the penultimate row in Fig. 9). In the remaining cases, *m*'s truth value is $\mathtt{True}$. It depends on *m*'s left child node if the node is with a False truth value (Case 2 and Case 4, and on *m*'s right child node if the node is with a $\mathtt{True}$ truth value (Case 2 and Case 3). This again exactly corresponds to MG's substantial conditions (the third-to-last row in Fig. 9). MG taints right those children that generate must-links.

Combining the base step and the inductive step, we prove that MG generates all must-links with no may-link. □

3.6 | Applying MG to Constraint Checking

MG minimizes the generated links during constraint checking. The improvement is generic and can be easily applied to existing constraint checking techniques, e.g., ECC (entire checking)[11], PCC (partial checking)[11], and Con-C (parallel checking)[23]. Hereafter, we apply a name convention that refers to different constraint checking techniques with different link generation techniques by their combination, e.g., ECC-CG for ECC equipped with CG (the original ECC), and ConC-MG for Con-C equipped with MG (MG replacing the original CG step in Con-C).

Applying MG to ECC and Con-C. This is straightforward. ECC and Con-C use an unmodified version of CG semantics listed in Fig. 5. MG can be directly applied by replacing the CG step with substantial derivation, conditional tainting, and S-CCT-guided link generation.

Applying MG to PCC. This use case deserves some discussion. PCC differs from ECC and Con-C in that it reuses previously calculated links instead of always regenerating them as in ECC and Con-C. Thus, it introduces a slight complexity. In each constraint checking turn, PCC does not destroy the CCTs inherited from previous turns but partially updates them. As a result,

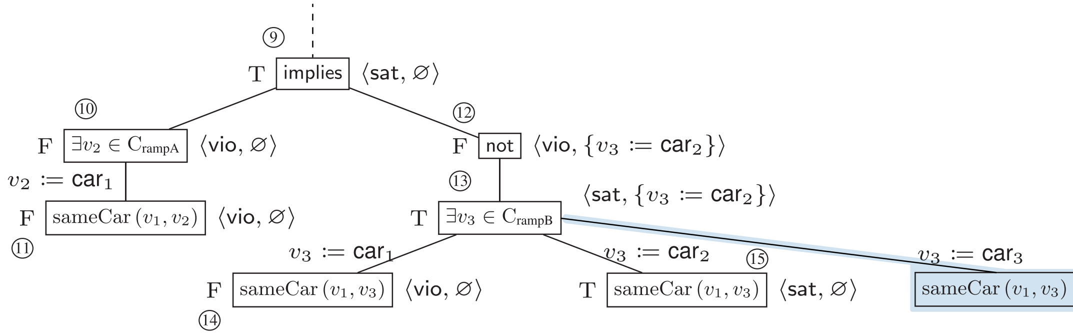


Figure 11 The modified sub-tree rooted for node 9 from Fig. 4

we have to consider two cases when applying MG to PCC: (1) for the updated part of a CCT, MG only generates links for tainted nodes; (2) for the reused part that is not updated, MG checks whether links on them are readily usable (not readily usable if the concerned nodes are not tainted in the last turn)—if so, it will reuse these links, and otherwise, it will regenerate links for those tainted nodes in the reused part. In other words, MG still generates must-links but additionally takes care of the reusing and deferred generation issues specific to PCC.

4 | REDUCED TRUTH VALUE EVALUATION BY S-CCTS

In this section, we exploit S-CCTs to improve the efficiency of truth value evaluation, another core step of constraint checking. It is an unexpected benefit that shows the potential of our S-CCT analysis. We use a motivating example to illustrate this optimization, followed by detailed explanations. MG equipped with this extra optimization constitutes MG+, a technique extended over MG that can further boost the efficiency of constraint checking.

4.1 | Motivating Example

An S-CCT taints all substantial parts contributing to the final constraint checking result. Besides link generation, truth value evaluation can also leverage such "necessary-versus-unnecessary" information to reduce redundancy. The key insight is that if a truth value is neither used by the current final result nor by any links generated later, it does not need to be evaluated.

Recall our example in Fig. 4 and focus on the right sub-tree rooted at Node 9 in Fig. 11. In our MG technique, the whole sub-tree is not tainted (i.e., not part of the S-CCT), so no link on it will be generated. Moreover, "not tainted" means that the sub-tree does not affect the final constraint checking result, including the truth value and the link on Node 1, so if later the sub-tree keeps this property, we can safely escape the evaluation of truth values on it too.

For example, suppose we add a new element car_3 to C_{rampB}, changing the sub-tree to sprout a new branch (the shadowed branch in Fig. 11). According to the truth value evaluation semantics of the existential formula (see Fig. 3), such a change will never alter the truth value of Node 13 and thus will keep the truth value of Node 9 unchanged. In this case, according to Alg. 1, the whole sub-tree stays untainted, meaning that there is still no node on the S-CCT. As a result, neither do links on the sub-tree require the truth values (MG avoids generating these links at all), nor do the final result require them (the truth value of Node 9 stays unchanged and thus "blocks" the influence of the newly added branch to the truth value in the final result). This means that skipping the evaluation of truth value on the sub-tree causes no harm. It shows that, though the original goal of S-CCT is to minimize link generation, it can simultaneously optimize another core step of constraint checking, too!

The insight extracted from the motivating example is that as long as sub-trees rooted at nodes like Node 9 (anchor nodes, explained later) stay untainted after a context change, we can safely escape all truth value evaluation on it.

We briefly introduce such reduction in the next section.

4.2 | Methodology Overview

Fig. 12 gives an overview of how MG+ works. It consists of three steps. First, MG+ catches all *anchor nodes* on the current CCT after a constraint checking turn according to the S-CCT. Anchor nodes are untainted nodes adjacent to the S-CCT, e.g., Node 8 and Node 9 in Fig. 4. Anchor nodes will be further examined by MG+ to find chances to escape the truth value evaluation on sub-trees rooted at them in the next checking turn. Second, based on the truth value of each anchor node, we derive escape conditions for them. An escape condition matches changes that keep the anchor node untainted. Also, it ensures the truth value of the anchor node is unchanging. Third, MG+ reduces the truth value evaluation by escape evaluating sub-trees rooted at anchor nodes whose escape conditions match the upcoming change. We will elaborate on these steps next.

4.3 | Catching Anchor Nodes

Anchor nodes are untainted nodes adjacent to the S-CCT. For example, considering the S-CCT in Fig. 4, Node 8 and Node 9 are anchor nodes. The parent node of an anchor node is exactly on the S-CCT. Recall the substantial conditions in Fig. 9. It is trivial that an anchor node cannot have a “not” parent since if a “not” node is tainted, its child must also be tainted (the last line in Fig. 9). Therefore, an anchor node must satisfy one of the following two cases:

- **The anchor node is a child of a ∀ or ∃ node.** In this case, the truth value of the anchor node must be True for the ∀ case or False for the ∃ case according to the substantial conditions. It is because the children of a ∀ node with a False truth values are always tainted if the ∀ node is on the S-CCT and thus cannot be anchor nodes. As long as the anchor node maintains the unchanged truth value, it stays untainted in the next checking turn. We can thus safely skip the sub-tree rooted at it in truth value evaluation.

- **The anchor node is a child of an “and,” “or” or “implies” node.** In this case, whatever the truth value of the parent node is, if both children of the parent node maintain unchanged truth values in the next checking turn, the anchor node stays untainted. This is because, currently, the anchor node is left out of the tainting process due to its breach of substantial conditions. This is caused by the truth values of the anchor node and its parent. In the next checking turn, since both the anchor node and the parent maintain unchanged truth values (unchanged truth values of the children lead to an unchanged truth value of the parent), they will keep the breach in the next checking turn, and the anchor node as a result stays untainted.

In practice, we can easily catch anchor nodes by recording boundary nodes of an S-CCT during MG’s analyses. Our goal is to escape unnecessary truth value evaluation that can be safely skipped. This should hold when an upcoming context change meets one of the above conditions. In this situation, we only need to cache the truth value of the anchor node. The number of anchor nodes is typically small since S-CCTs typically occupy a small part on the original CCT. Thus, the overhead is negligible. The correctness of escaping such evaluation is guaranteed by the following theorem.

Theorem 3. Any anchor node with a ∀ or ∃ parent stays untainted upon an upcoming context change if it maintains an unchanged truth value; any anchor node with an “and” or “or” or “implies” parent stays untainted upon an upcoming context change if it and its sibling maintain unchanged truth values.

Sketch of proof: The proof formally echoes our previous intuitive explanation. Suppose we have an anchor node n whose parent is p. We have to prove the theorem respecting two cases.

***p* is a ∀ node.** According to the substantial conditions (Fig. 9) and tainting algorithm (Alg. 1), the only possible truth value of p is True and the only possible truth value of n is False since p is on the S-CCT but n is not (Line 1 in Fig. 9). Consider

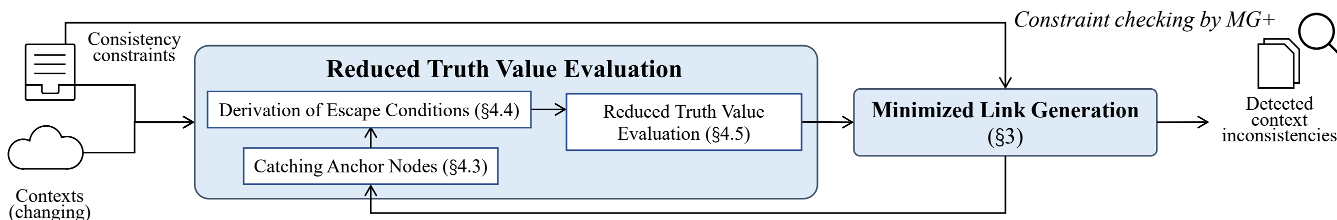


Figure 12 MG+ overview

any upcoming context change. The change may alter some truth values on the CCT, which may or may not keep p tainted. If p becomes untainted, it is naturally excluded from the S-CCT. Otherwise, there are four possible cases on how p's truth value changes: (1) staying True, (2) from True to False, (3) from False to True, and (4) staying False. In all four cases, according to the substantial conditions respecting ∀ formulas, viz., $(\forall, \mathrm{True}) \xrightarrow{SC} \emptyset$ and $(\forall, \mathrm{False} \xrightarrow{SC} \ulcorner >, \mathrm{False} \urcorner$, n always stays untainted if its truth value stays to be True (so that it can never meet the substantial conditions). The proof when p is an ∃ node is similar.

p is an "and" node. According to the substantial conditions (Fig. 9) and tainting algorithm (Alg. 1), the only possible truth value of n is True and the only possible truth value of n's sibling is False. They together result in a False truth value of p. After the context change, we request that the truth values of n and n's sibling both stay unchanged. Again, p may or may not stay untainted, and if it becomes tainted, n is naturally excluded from the S-CCT. Otherwise, since the truth values of p, n, and n's sibling stay to be False, True, and False, n stays untainted. The proof when p is an "or" or "implies" node is similar.

Combining all these cases, we conclude our proof. □

4.4 | Derivation of Escape Conditions

In this step, we propose a lightweight method to examine whether a context change would alter the truth values of an anchor node and (if necessary) its siblings. With this method, we can quickly decide whether a context change would require a re-evaluation of a sub-tree rooted at any anchor node. If not, the sub-tree can be safely escaped from the truth value evaluation step (detailed in the next section).

Escape conditions for an anchor node are patterns that match changes that keep the anchor node untainted. If a change meets any escape condition, we can escape re-evaluation of truth values on the sub-tree rooted in it. As discussed before, escape conditions of anchor nodes with ∀ or ∃ parents only need to keep the truth value of the anchor node itself unchanged and escape conditions of anchor nodes with "and," "or," or "implies" parents should keep the truth value of its sibling unchanged too.

We derive escape conditions following a two-step analysis.

In the first step, we statically analyze which context changes can keep the current truth value of an anchor node. This is possible because the analysis only depends on the static formula structure of the sub-tree rooted at that anchor node. Take a simple constraint $\forall v \in \mathrm{C}\,(bfunc)$ as an example. After adding an element to C, the only possible change of the truth value is from True to False, and if the current truth value is already False, it stays unchanged. Similarly, after deleting an element from C, the only possible change of the truth value is from False to True, and if the current truth value is already True, it stays unchanged. Thus, we can obtain two sets, $\mathrm{ECondition_F}$ and $\mathrm{ECondition_T}$, for it, where $\mathrm{ECondition_F}$ matches context changes that keep the current truth value unchanged if it is already False, and $\mathrm{ECondition_T}$ matches the opposite. The value of $\mathrm{ECondition_F}$ in this example is $\{\langle +, \mathrm{C}\}$ and the value of $\mathrm{ECondition_T}$ is $\{\langle -, \mathrm{C}\}$. For more complex formulas, we can similarly obtain these two sets recursively, following derivation rules in Fig. 13.

In the second step, we synthesize complete escape conditions from $\mathrm{EConditions_T}$ and $\mathrm{EConditions_F}$, following derivation rules in Fig. 14. Specifically, if the current truth value of an anchor node is True, we choose $\mathrm{EConditions_T}$ to keep that truth

$$
\begin{aligned}
&\mathcal{E}'_{\mathrm{T}}\left[(f_1)\ \text{and}\ (f_2)\right] = \mathcal{E}'_{\mathrm{T}}\left[f_1\right] \cup \mathcal{E}_{\mathrm{T'}}\left[f_2\right]\\
&\mathcal{E}'_{\mathrm{T}}\left[(f_1)\ \text{or}\ (f_2)\right] = \mathcal{E}'_{\mathrm{T}}\left[f_1\right] \cup \mathcal{E}'_{\mathrm{T}}\left[f_2\right]\\
&\mathcal{E}'_{\mathrm{T}}\left[(f_1)\ \text{implies}\ (f_2)\right] = \mathcal{E}'_{\mathrm{F}}\left[f_1\right] \cup \mathcal{E}'_{\mathrm{T}}\left[f_2\right]\\
&\mathcal{E}'_{\mathrm{T}}\left[\text{not}\ (f)\right] = \mathcal{E}'_{\mathrm{F}}\left[f\right]\\
&\mathcal{E}'_{\mathrm{T}}\left[\forall v \in \mathrm{C}\,(f)\right] = \mathcal{E}'_{\mathrm{T}}\left[f\right] \cup \{\langle -, \mathrm{C}\rangle\}\\
&\mathcal{E}'_{\mathrm{T}}\left[\exists v \in \mathrm{C}\,(f)\right] = \mathcal{E}'_{\mathrm{T}}\left[f\right] \cup \{\langle +, \mathrm{C}\rangle\}\\
&\mathcal{E}'_{\mathrm{T}}\left[bfunc\left(v_1, ..., v_n\right)\right] = \emptyset
\end{aligned}
$$

(a) Derivation rules for set $\mathrm{EConditions_T}$

$$
\begin{aligned}
&\mathcal{E}'_{\mathrm{F}}\left[(f_1)\ \text{and}\ (f_2)\right] = \mathcal{E}'_{\mathrm{F}}\left[f_1\right] \cup \mathcal{E}'_{\mathrm{F}}\left[f_2\right]\\
&\mathcal{E}'_{\mathrm{F}}\left[(f_1)\ \text{or}\ (f_2)\right] = \mathcal{E}'_{\mathrm{F}}\left[f_1\right] \cup \mathcal{E}'_{\mathrm{F}}\left[f_2\right]\\
&\mathcal{E}'_{\mathrm{F}}\left[(f_1)\ \text{implies}\ (f_2)\right] = \mathcal{E}'_{\mathrm{T}}\left[f_1\right] \cup \mathcal{E}'_{\mathrm{F}}\left[f_2\right]\\
&\mathcal{E}'_{\mathrm{F}}\left[\text{not}\ (f)\right] = \mathcal{E}'_{\mathrm{T}}\left[f\right]\\
&\mathcal{E}'_{\mathrm{F}}\left[\forall v \in \mathrm{C}\,(f)\right] = \mathcal{E}'_{\mathrm{F}}\left[f\right] \cup \{\langle +, \mathrm{C}\rangle\}\\
&\mathcal{E}'_{\mathrm{F}}\left[\exists v \in \mathrm{C}\,(f)\right] = \mathcal{E}'_{\mathrm{F}}\left[f\right] \cup \{\langle -, \mathrm{C}\rangle\}\\
&\mathcal{E}'_{\mathrm{F}}\left[bfunc\left(v_1, ..., v_n\right)\right] = \emptyset
\end{aligned}
$$

(b) Derivation rules for set $\mathrm{EConditions_F}$

Figure 13 Derivation rules for partial escape condition sets of different formulas

$\mathcal{E}_{\mathrm{T}}[f] =$
(1) $\mathcal{E}'_{\mathrm{T}}[f] \cup \mathcal{E}'_{\mathrm{F}}[f']$,
when (f) and (f') or (f') and (f)
(2) $\mathcal{E}'_{\mathrm{T}}[f] \cup \mathcal{E}'_{\mathrm{T}}[f']$,
when (f) implies (f')
(3) $\mathcal{E}'_{\mathrm{T}}[f]$, otherwise

(a) E. Conds for True

$\mathcal{E}_{\mathrm{F}}[f] =$
(1) $\mathcal{E}'_{\mathrm{F}}[f] \cup \mathcal{E}'_{\mathrm{T}}[f']$,
when (f) or (f') or (f) or (f')
(2) $\mathcal{E}'_{\mathrm{F}}[f] \cup \mathcal{E}'_{\mathrm{F}}[f']$,
when (f') implies (f)
(3) $\mathcal{E}'_{\mathrm{F}}[f]$, otherwise

(b) E. Conds for False

$\mathcal{E}[f]_{\alpha} =$
(1) $\mathcal{E}_{\mathrm{T}}[f]$, if $\mathcal{T}[f]_{\alpha} = \mathrm{T}$
(2) $\mathcal{E}_{\mathrm{F}}[f]$, if $\mathcal{F}[f]_{\alpha} = \mathrm{F}$

(c) E. Conds at runtime

Figure 14 Derivation rules for escape conditions of different anchor nodes

value unchanged, and vice versa. Moreover, for anchor nodes with an "and," "or" or "implies" parent, we should also consider its sibling. For example, if the truth value of an anchor node *n* is True, we should include (1) escape conditions that keep this truth value unchanged (EConditions$_{\mathrm{T}}$ of *n*), and (2) escape conditions that keep the truth value of *n*'s sibling (which must be False) unchanged (EConditions$_{\mathrm{F}}$ of *n*'s sibling). We use the union instead of the intersection because one change may only affect one sub-formula. If it affects the sub-formula of *n*, it will not affect the sub-formula of *n*'s sibling, leaving all truth values on that sub-tree rooted unchanged. Thus, if the change affects *n* and meets one of *n*'s escape conditions, it implicitly meets the escape conditions of *n*'s sibling too.

Note that since the first step is static, we can perform it in advance, completely avoiding runtime overheads. Besides, the second step consists of simple assigning and set operations, which are quite efficient.

Theorem 4 proves the correctness of derived escape conditions.

Theorem 4. Context changes satisfying one of the escape conditions of an anchor node will keep the truth value of that anchor node unchanged, and if the anchor node has an "and," "or," or "implies" parent, it will keep the truth value of the anchor node's sibling unchanged too.

Sketch of proof: Consider an anchor node *m*. It corresponds to the formula f. The upcoming context change directly modifies a context in the sub-tree rooted at *m*. Suppose the context is associated with a universal or existential node *n*. We first prove that if *m*'s current truth value is True and chg conforms to $\mathcal{E}'_{\mathrm{T}}[f]$, the truth value will stay to be True. Similarly, if *m*'s old truth value is False and chg conforms to $\mathcal{E}'_{\mathrm{F}}[f]$, the truth value will stay to be False. We achieve this by induction over the length d of the path from *m* to n.

Base case Let d be 0. It indicates that *m* and *n* are the same node. In this case, *m* itself is a '∀' or "∃" node directly affected by this change. Suppose that f is $\forall v \in \mathrm{C}\left(f'\right)$. When *m*'s current truth value is True, according to the derivation rules of escape conditions (Fig. 13 and Fig. 14), the context change must be a deletion change to C (otherwise it cannot meet these escape conditions). Since *m*'s truth value is True, all its children must be with True truth values. After deleting one of its children, the remaining children are still with True truth values. This keeps *m*'s True truth value. If there is no child left, *m*'s default truth value is also True. When *m*'s old truth value is False, according to the derivation rules of escape conditions (Fig. 13 and Fig. 14) the context change must be an addition change to C. Also, there should be a child of *m* with a False truth value, and the addition change will not affect that child. Thus, *m* keeps its False truth value. The analysis is similar when *m* is an ∃ node.

Inductive step Let d be a number h greater than 0. Assume that when d is $h-1$, this property holds. Let *p* be *m*'s child node on the path from *m* to n. There are $h-1$ nodes on the path from *p* to n. We discuss different situations respecting m's formula type. For conciseness, only discuss cases for "and," "not," and ∀ formulas.

***m* is an "and" node** Let f be $\left(f_1\right)$ and $\left(f_2\right)$ and *p* corresponds to f_1. When *m*'s current truth value is True, the context change must satisfy some condition in $\mathcal{E}_{\mathrm{T}}\left(f_1\right)$. Otherwise, it should satisfy some condition in $\mathcal{E}_{\mathrm{T}}\left(f_2\right)$ that is irrelevant with *p*'s sub-tree. This contradicts the fact that *n* is a descendant of *p*, so is impossible. Also, *p*'s truth value should be True (only when both children with True truth values, an "and" formula can be evaluated to True). According to the inductive hypothesis, *p*'s truth value will stay True after applying the change. The truth value of *p*'s sibling will stay to be True too, because the context change does not affect any of its descendants at all. Thus,

m's truth value stays unchanged. When m's truth value is False, the context change must satisfy some condition in $\mathcal{E}_{\mathrm{F}}\left(f_1\right)$. In this case, p's current truth value can be either True or False. p with a True truth value means that p's sibling is with a False truth value. The False truth value itself ensures that m's truth value stays to be False after the change. When p is with a False truth value, by the inductive hypothesis, its truth value after the change stays to be False, and m's truth value thus keeps being False too.

***m* is a "not" node** Let f be $\mathsf{not}\left(f'\right)$ and p corresponds to f'. When m's current truth value is True, the context change must satisfy one condition in $\mathcal{E}_{\mathrm{F}}\left(f'\right)$. The only possible truth value of p is False, and the inductive hypothesis ensures that it stays unchanged. This, in turn, keeps m's truth value unchanged. The analysis is similar when m's truth value is False.

***m* is a ∀ node** Let f be $\forall v \in \mathrm{C}\left(f'\right)$ and p corresponds to f'. When m's truth value is True, the context change must satisfy some condition in $\mathcal{E}_{\mathrm{T}}\left(f'\right)$ because it does not directly modify C (so cannot be $\langle +/-, \mathrm{C}\rangle$). p's current truth value must be True and the inductive hypothesis keeps it unchanged. When m's truth value is False, the context change must satisfy some condition in $\mathcal{E}_{\mathrm{F}}\left(f'\right)$. If p's current truth value is True, p's some sibling must be with a False truth value, and the change does not affect that sibling at all, so the sibling's truth value will keep m's truth value being False. If p's current truth value is False, the inductive hypothesis directly ensures that this truth value will not change.

Hereafter, m's parent node is l. We proceed to prove that if m is with a True truth value, $\mathcal{E}_{\mathrm{F}}(f)$ keeps that truth value unchanged, and if l is an "and," "or," or "implies" node, the truth value of m's sibling keeps unchanged too. We analyze the three cases in the computation of $\mathcal{E}_{\mathrm{T}}[f]$ in Fig. 14b respectively.

1. ***l* is an "and" node.** m's sibling must be with a False truth value or otherwise m cannot be an anchor node. In this case, the context change that conforms to $\mathcal{E}_{\mathrm{T}}[f]$ should be in either $\mathcal{E}'_{\mathrm{T}}[f]$ or $\mathcal{E}'_{\mathrm{F}}\left[f'\right]$. In both cases, the previous proved property of $\mathcal{E}'_{\mathrm{T}}[f]$ and $\mathcal{E}'_{\mathrm{F}}\left[f'\right]$ keeps the two truth values of m and m's sibling unchanged.

2. ***l* is an "implies" node and *m* is its left child.** m's sibling must be with a True truth value. In this case, the context change that conforms to $\mathcal{E}_{\mathrm{T}}[f]$ should be in either $\mathcal{E}'_{\mathrm{T}}[f]$ or $\mathcal{E}'_{\mathrm{T}}\left[f'\right]$. Again, the previous proved property of $\mathcal{E}'_{\mathrm{T}}[f]$ and $\mathcal{E}'_{\mathrm{F}}\left[f'\right]$ keeps the two truth values of m and m's sibling unchanged.

3. **Other cases where *m* is with a True truth value.** It is impossible for l to be an "or" node with a True truth value while m is an anchor node, so l is either a ∀ node or an ∃ node. In these cases, $\mathcal{E}_{\mathrm{T}}[f]$ is just $\mathcal{E}'_{\mathrm{T}}[f]$. The previously proved property of $\mathcal{E}'_{\mathrm{T}}[f]$ keeps m's truth value unchanged.

We can similarly prove such a property for $\mathcal{E}_{\mathrm{F}}[f]$.

Eventually, the computation of $\mathcal{E}[f]_\alpha$ is just picking up $\mathcal{E}_{\mathrm{T}}[f]$ or $\mathcal{E}_{\mathrm{F}}[f]$ according to m's truth value at runtime. When the truth value is True, the above property of $\mathcal{E}_{\mathrm{T}}[f]$ keeps m's truth value unchanged after applying the context change, and when the truth value is False, the property of $\mathcal{E}_{\mathrm{F}}[f]$ covers the opposite case.

Combining all the above steps, we conclude the proof. □

With Theorems 3 and Theorem 4, we can guarantee an anchor node to be untainted after applying a context change that meets any escape condition. Therefore, we can safely escape truth value evaluation for the sub-tree rooted at it because the re-evaluated truth values will not affect the final result at all.

4.5 | Reduced Truth Value Evaluation based on Escape Conditions

Given anchor nodes and their escape conditions, we can reduce unnecessary truth value evaluation. As mentioned before in Section 2.2.1, the truth value evaluation is conducted in a bottom-up manner, i.e., by a post-order traversal: a node is evaluated only after all its children have been evaluated. Our reduced truth value evaluation provides a fast path for this process.

Alg. 2 depicts the algorithm. It recursively evaluates the whole CCT as usual until the context change meets the escape conditions of some anchor node (Line 5 and Line 6). If that happens, the fast path directly returns the previously evaluated truth value (Line 7).

Algorithm 2 Reduced Truth Value Evaluation based on Escape Conditions

```
1: procedure EVALUATE(cct, chg)
2:     return EVALUATENODE(cct.root, chg)
3: end procedure
4: procedure EVALUATENODE(currentNode, chg)
5:     if currentNode.isAnchor() then
6:         if MATCH(currentNode.getEConditions(), chg) then
7:             return currentNode.oldTruthValue
8:         end if
9:     else
10:        return ORIGINEVALUATE(currentNode, chg)
11:    end if
12: end procedure
```

4.6 | Applying MG+ to Constraint Checking

MG+ refines MG by using the S-CCT information to reduce unnecessary truth value evaluation. This optimization is generic and can be easily applied to existing constraint checking techniques, e.g., ECC (entire checking)[11], PCC (partial checking)[11], and Con-C (concurrent checking)[23], too. We discuss these use cases in the following.

Applying MG+ to ECC and Con-C. This is straightforward. Following the aforementioned three steps, MG+ records anchor nodes after each checking turn, analyzes their escape conditions, and conducts reduced truth value evaluation. The only difference is that since ECC and Con-C do not save CCT states, MG+ needs to maintain some necessary information, like previous truth values of anchor nodes, by itself. The time and space cost of such information is reasonable and controllable since the number of anchor nodes is limited.

Applying MG+ to PCC. This use case deserves some discussion. PCC may reuse truth values of CCT nodes evaluated in previous checking turns, and these truth values have to be updated in a timely manner before being used to generate links. Simply escaping truth value evaluation on a sub-tree may cause problems in the subsequent checking process. Thus, we restrict when we can apply the optimization for PCC. It is conducted only when the anchor node is $\forall$ or $\exists$ and the context change directly modifies the context related to it. Otherwise, we follow PCC's original truth value evaluation algorithm to control MG+'s possible influence on PCC's incremental mechanisms.

5 | EVALUATION

In this section, we evaluation and compare MG and MG+ to existing work on their effectiveness on minimizing link generation and reducing truth value evaluation, and their actual improvement in the efficiency of constraint checking.

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

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

RQ4 (MG+ Effectiveness) How effective is the MG+ in reducing truth value evaluation, as compared to the original one in constraint checking?

RQ5 (MG+ Benefits) How does MG+'s reduced truth value evaluation contribute to the efficiency improvement of existing constraint checking (with ECC, PCC, and Con-C)?

5.2 | Experimental Design and Setup

We answer RQ1–RQ2 by a comparative study with exhaustively synthesized constraints and controlled factors and answer RQ3–RQ5 by a case study with real-world data. We explain the evaluation design and setup below.

5.2.1 | RQ1 and RQ2

We study the characteristics of different link generation techniques on link redundancy. We use synthesized constraints to conduct the study to explore all possible formula structures. The synthesis is briefly an exhaustive enumeration process. For example, there is one type of 1-layer formula: $bfunc$, and to construct 2-layer formulas, we traverse all 1-layer formulas and combine them using logical conjunctions ("and," "or," "implies," and "not") and quantifiers (∀ and ∃), and so on. An example of these synthesized formulas is the following 3-layer one:

$$\forall v \in C\,((bfunc)\ \texttt{and}\ (bfunc))$$

Obviously, there must be a height limit for the synthesized constraints, or otherwise, we will get infinite ones. We set the limit to 4 and get 1,658 well-formed constraints. We choose this limit due to two reasons: (1) 1,658 is already a sufficient number that includes various constraints, covering enough kinds of formula structures for our analysis; (2) increasing the height limit to 5 will drastically add over 10^8 more constraints, and that overwhelms computational capability of any computer.

With these synthesized constraints, we decide how to calculate $bfunc$ values which do not carry real semantics in a synthesized formula. Though, one can simulate the calculation of their values by two parameters: (1) the number l of elements in the constraint contexts, and (2) the probability p of whether a $bfunc$ returns True or False. For example, if we set l to 5 and p to 0.05, there will be five branches at runtime in the CCT constructed from the constraint $\forall v \in \mathrm{C}\,(bfunc)$, and each leaf (a $bfunc$) returns True with a 0.05 probability. We test l set to be 2, 5, 10, 15, and 20, and p set to be 0.01, 0.05, 0.1, 0.3, 0.5, 0.7, 0.9, 0.95, and 0.99 to cover different scenes of constraint checking. 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.

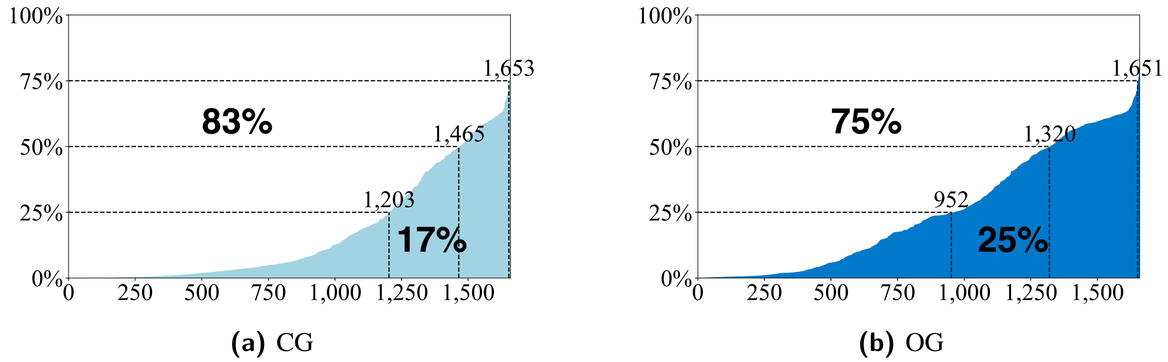
We use *link utilization rates* (ULRs) to reveal the severity of link redundancy. A ULR is computed by dividing the number of actually used links by the number of total links. A higher ULR means less link redundancy and a 100% ULR means absolutely no redundancy (all links are actually used). To answer RQ1, we compare the ULRs of CG and OG on different constraints. We calculate their average to show how severe the problem is and their ranges to show the heterogeneity of these constraints. Also, we calculate the standard deviation of ULRs when applying CG and OG to check the stability of the performance of these techniques. To answer RQ1, we assess the ULRs after applying MG (MG+ has no difference with MG on link redundancy) to explore its improvement over CG and OG.

5.2.2 | RQ3–RQ5

We follow existing work [11,23,24,25] to use a real-world application, SmartCity, with its large-volume taxi-driving data in the case study to evaluate the actual impact of MG and MG+ on the efficiency of performance. Also, in the case study, we assess the effectiveness of MG+ on reducing truth value evaluation. Since truth values must hold real semantics to precisely assess MG+, we cannot use the aforementioned synthesized constraints to do that.

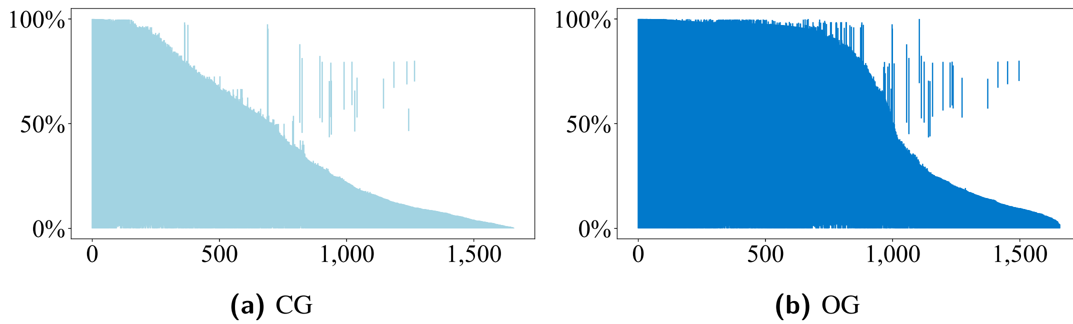
The SmartCity application is used by the transportation department of a city in southern China to manage and guide taxi driving. The data are heavy, and they change frequently due to vehicles' quick motions and complex behaviors (e.g., turning, picking up a passenger, and going off duty). The staff collected a total of 1.55 million raw taxi data covering 760 vehicles within a continuous period of 24 hours (as sample data for research purposes). The data were transformed to 6.75 million context changes, with respect to the application's deployed 22 consistency constraints (for continually guarding the correctness of the data used by the smart guidance functionalities in the application). These constraints cover different aspects of vehicles' movements with respect to their surrounding environments, e.g., speed limit, highway layouts, driving direction, etc.

We fed these data and constraints to constraint checking techniques with CG, OG, and MG, respectively, and counted (1) the number of evaluated truth values to assess how effective MG+ is in reducing truth value evaluation (RQ4) and (2) the time of link generation, truth value evaluation, and the whole checking process to assess the actual impacts of MG and MG+ on the



(a) CG **(b)** OG

Figure 15 Averaged ULRs for CG and OG



(a) CG **(b)** OG

Figure 16 ULR ranges for CG and OG

efficiency of constraint checking (RQ3 and RQ5). A small number of evaluated truth values will mean that MG+ successfully reduces much truth value evaluation, and a short checking time will mean MG/MG+ improves the efficiency of constraint checking. These three RQs (3, 4, and 5) were evaluated in a case study setting where one checked massive data from the above real-world SmartCity application with 6.75 million context changes. Therefore, we conducted the corresponding experiments by running and collecting the time of different steps and also the total time of constraint checking for completing all data checking and outputting detected inconsistencies.

5.2.3 | Implementation and the Running Environment

We implement our method in Java. The code and data with a document about how to run our implementation are released on GitHub[32]. 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.

5.3 | Result Analyses

5.3.1 | RQ1 (Motivation)

Fig. 15 illustrates the averaged ULR measures for CG and OG across all 1,658 consistency constraints. In the figure, each line is the ULR of a constraint (the same for Fig. 16 and Fig. 17 after). 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. 15, 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

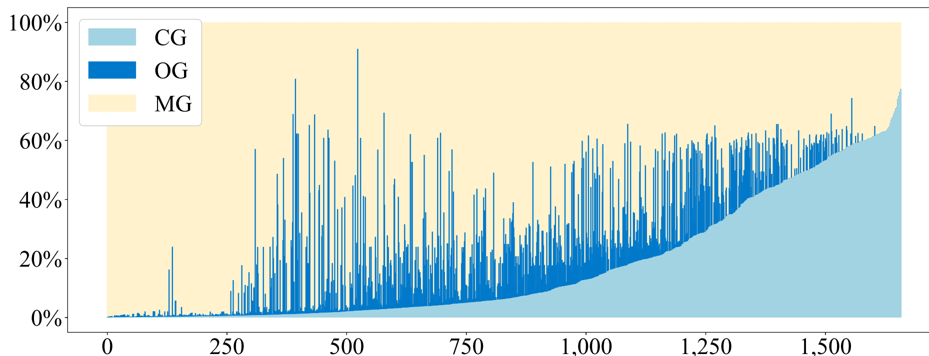


Figure 17 ULR comparison for CG, OG, and MG

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. 16, 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. These data echoes the standard deviation of different constraints, 21.52% for CG and 23.26% for OG, which are quite high for a ratio ranging from 0 to 100%. Note that OG is even more unstable than CG, for it has wider ULR ranges (see Fig. 16) and a larger standard diviation. 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 significantly affected by the issue of link redundancy, underscoring the urgent need for new research to address this problem. 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.

5.3.2 | RQ2 (MG Effectiveness)

Fig. 17 compares the averaged ULR measures for CG, OG, and MG across all 1,658 constraints. This gives an intuitive and exhaustive picture of 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.

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 significant.

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

5.3.3 | RQ3 (MG Benefits)

Fig. 18 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.

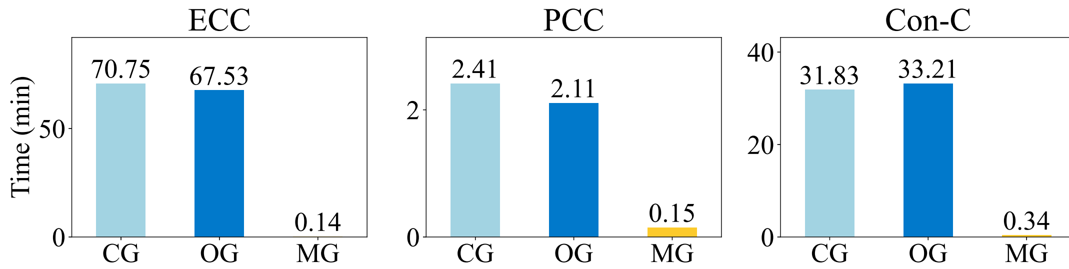


Figure 18 Link generation time comparison for CG, OG, and MG

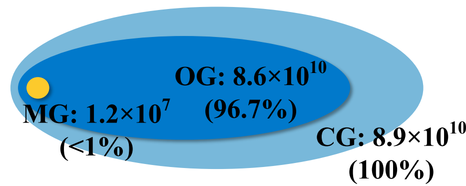


Figure 19 Illustration of generated links for CG, OG, and MG

Table 1 Comparisons on truth value calculations between MG and MG+

Technique	MG's Calculations (#)	MG+'s Calculations (#)	Reduction (%)
ECC/Con-C	8.911E10	4.479E10	49.74%
PCC	8.911E10	8.909E10	0.02%

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. 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).

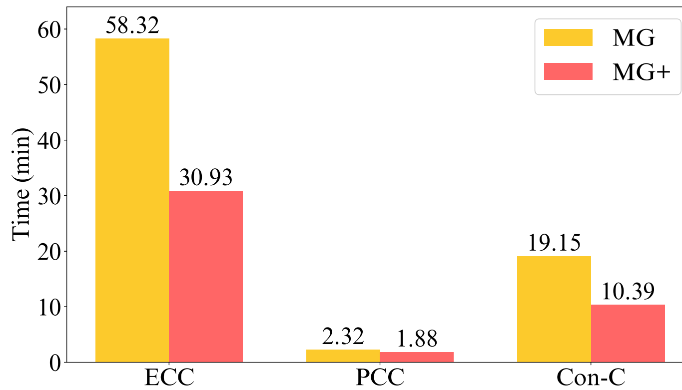


Figure 20 Truth value evaluation time comparison for MG and MG+

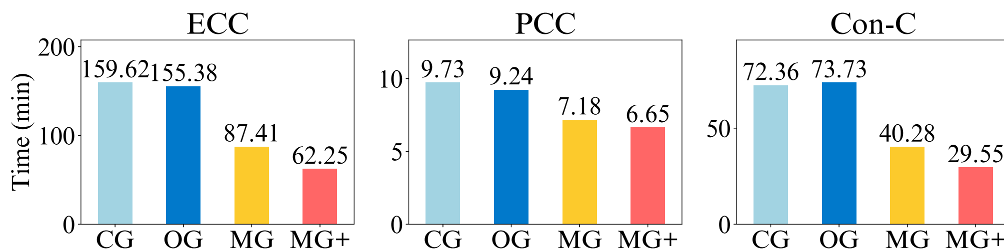


Figure 21 Total checking time comparison for MG and MG+

5.3.4 | RQ4 (MG+ Effectiveness)

Table 1 compares the number of calculated truth values between MG and MG+, combined with ECC, PCC, and Con-C under the case study scenario. Note that when MG is adopted, this means traditional truth value evaluation is used and all nodes' truth values need to be evaluated during the whole checking, either calculating from scratch like ECC and Con-C, or inheriting the latest values from the checking history like PCC.

From the table, MG evaluates a total of 8.911×10^{10} truth values during the whole checking under the real-world application scenario with 22 consistency constraints and 6.75 million context data. However, when MG+ is adopted, some of truth values are escaped from any calculation due to MG+'s escape condition analyses. Indeed, MG+'s reduced truth value evaluation actually alleviates the workloads of truth value evaluation in constraint checking. To be specific, MG+ only needs to evaluate 4.479×10^{10} truth values in total, successfully reducing around 49.74% calculations for ECC and Con-C. When MG+ is applied to PCC, as we discussed in Section 4.6, in order to preserve PCC's incremental mechanism, MG+'s reduction is restricted therefore some escapable truth values would still be calculated in case of their potential usages in future for PCC. Therefore, MG+ only reduces 0.02% truth value calculations for PCC, still smaller than those of MG. Based on obvious reductions in truth value evaluation, we expect that MG+ can also improve MG's efficiency in constraint checking.

Therefore, we conclude that MG+ can significantly reduce around half of truth value calculations for ECC and Con-C. Although for PCC, the reduction is small, it is still positive.

5.3.5 | RQ5 (MG+ Benefits)

Fig. 20 compares the truth value evaluation time between MG and MG+, when they are combined with ECC, PCC, and Con-C. From the figure, we observe that when combined with ECC and Con-C, MG+ reduces 45.74% and 47.0% time cost in the truth value evaluation. This echoes the reduction rate of calculated truth values as we discussed in RQ4. Besides, although MG+'s reduction on calculated truth values is relatively marginal (0.02%), we still observe that MG+ can achieve 19.0% checking time reduction as compared to MG, when PCC is adopted. This is because, when applying MG+ to PCC, we emphasize reducing

truth value calculations, especially to ∀ and ∃ anchor nodes. Although these nodes' reduced truth values may be small in number, they usually occupy the most time costs during the evaluation due to their complex logics.

Note that most computations of MG+ algorithms are static and can be conducted ahead of time (i.e., not at runtime). Therefore, the runtime overheads were only less than hundreds of milliseconds, as compared to the total checking time at the level of several hours. Therefore, we believe that all the above improvements are achieved over MG+'s internal overhead, which is negligible. So we did not explicitly include it in the results.

Concerning the total checking time, including both truth value evaluation and link generation steps, we also investigate how MG+ (targeting both steps) and MG (targeting the link generation step) actually improve the checking efficiency individually. Fig. 21 gives the total checking time when CG, OG, MG, MG+ are combined for ECC, PCC, and Con-C. We can observe that MG's minimized link generation can help reduce 26.2–45.2% time cost over CG for the whole constraint checking, and 22.3–45.4% time cost over OG. As compared to MG, MG+'s additional reduced truth value evaluation can further reduce 7.4–28.2% time cost over MG for the whole constraint checking. To sum up, MG+ can reduce 31.7–61.0% time cost over CG for the whole constraint checking, and 28.0–60.0% time cost over OG.

Therefore, we conclude that by reducing quite a number of truth value calculations during the evaluation, MG+ can achieve obvious efficiency improvements against MG by additionally saving 19.0–47.0% time for truth value evaluation, thus together saving 7.4–28.2% time for the whole constraint checking as compared to MG.

5.4 | Threats to Validity

We analyze the construct validity, internal validity, and external validity of our evaluation.

5.4.1 | Construct Validity

We measure the effectiveness of CG, OG, MG, and MG+ by their link utilization rates, numbers of evaluated truth values, and checking time. These metrics are complete, and can directly reflect the performance of different techniques.

Validity of link utilization rates. We measured link utilization rates for CG, OG, MG/MG+ to assess the effectiveness of MG/MG+ on eliminating link redundancy. Link utilization rates are the ratio of actually used links to all links. Obviously, a higher link utilization rate means lower redundancy, and a 100% link utilization rate means no redundancy. Link utilization rates can directly reflect the severity of the link redundancy problem when applying a link generation technique.

Validity of numbers of evaluated truth values. We measured the numbers of evaluated truth values of CG, OG, MG, and MG+ combined with different constraint checking techniques (ECC, PCC, and Con-C) to assess the effectiveness of MG+ on reducing truth value evaluation. Similarly, a smaller number of evaluated truth value means fewer calculations performed during the truth value evaluation step, and a good truth value evaluation reduction technique of course produces fewer truth values. The value directly reflects the performance of the reduction.

Validity of checking time. We measured the checking time of CG, OG, MG, and MG+ combined with different constraint checking techniques (ECC, PCC, and Con-C) to assess the actual impacts of MG and MG+'s optimizations. The total checking time constructs of three parts, viz., constructing and maintaining CCTs, truth value evaluation, and link generation. The partition is complete since all constraint checking techniques have to build up CCTs, evaluate truth values on them, and generate links according to the truth values whatever the implementation is. Other time costs (e.g., launching a Java Virtual Machine) are specific to implementations and may vary. They are irrelevant to constraint checking. To evaluate MG and MG+, we specially analyze the link generation time (for minimized link generation), the truth value evaluation time (for reduced truth value evaluation), and the total time (for overall performance). The step of constructing and maintaining CCTs is irrelevant to the optimizations of MG and MG+ and stays identical in all compared techniques (CG, OG, MG, and MG+), so we did not explicitly analyze this part of the breakdown.

5.4.2 | Internal Validity

The internal validity may be affected by implementation bias and bugs.

Alleviating implementation bias. To avoid implementation bias, we re-implement all existing techniques within the same code base of MG and MG+. They share the same underlying data structures and algorithms, and the only difference is the constraint checking mechanism (truth value evaluation and link generation). This ensure fair play as much as possible.

Avoiding bugs. We use differential testing and metamorphic testing to avoid bugs to our best. The scenario of constraint checking are intrinsically haunted by the oracle problem[33,34,11,23,24,25,35]. The complexity and dynamicity of the data under checking makes it impossible to directly give known-to-be-correct results to validate our implementation. To address this problem, we conduct differential testing, i.e., testing the code by comparing results from different implementations (our implementation and implementation from previous works) and ensuring their consistency. Also, we conduct metamorphic testing using tools from literature[35], i.e., testing the code by constructing multiple inputs in a way that the corresponding outputs must conforms to some relations, and feeding these inputs to our implementation to check whether any relation is violated. We did not find any bug in our implementation insofar as differential testing and metamorphic testing can guarantee the correctness.

5.4.3 | External Validity

Possible threats to the external validity of our evaluation are (1) whether it can cover all possible situations that our techniques may encounter (exhaustiveness), and (2) whether it can reflect the actual performance of our techniques in the real world (actuality).

Exhaustiveness. There are two aspects of exhaustiveness. First, MG and MG+ may be applied to different constraint checking techniques. We selected representatives for existing research lines, viz., ECC as the baseline, PCC for incremental checking, and Con-C for parallel checking to ensure this aspect. In the evaluation, we combined MG and MG+ with these representatives respectively to evaluate their performance. Second, MG and MG+ have to handle different types of constraints. It is impossible to completely fulfill this aspect of exhaustiveness because of the infinite number of all possible constraints. However, we try our best by synthesizing all well-formed constraints not higher than 4 layers and conduct simulated experiments on them under different parameters in a comparative study. There are 1,658 such constraints. It is a sufficient number to draw statistically significant conclusion on the effectiveness of our methods, and extending the height limit to 5 brought us over 10^8 more constraints, which obviously exceeds ability of any computer to perform experiments.

Actuality. To ensure that our evaluation can reflect the actual performance of our techniques in the real world, we conduct a case study on a real-world SmartCity application with constraints written by their programmers and millions of context data collected by sensors in the system. The case study obtained consistent results, echoing our earlier comparative study.

6 | RELATED WORK

Our work improves the efficiency of constraint checking, and constraint checking extends two research lines in software engineering, viz., the oracle problem and the constraint solving techniques. Moreover, our ideas are inspired by works that reduce redundant computation to boost software efficiency in other fields. We proceed to elaborate these related topics one by one.

6.1 | Consistency Management and Constraint Checking

Software artifacts that evolve over time may encounter abnormal behaviors, i.e., inconsistencies, caused by broken code or unexpected scenarios. Managing the consistency of such software artifacts facilitates the delivery of adaptive and smart services by the associate applications.

Traditional software artifacts evolve slowly. Consistency management for these artifacts focuses on reliability, i.e., correct and complete detection of any inconsistencies. There is much consistency management literature for such artifacts, including XML documents[9,36,37,38], UML models[39,40,41], set-and-relation-based models[42], workflows[43,44], and distributed algorithms[45].

However, emerging software artifacts with high complexity and dynamicity require consistency management with extremely high efficiency, and previous techniques for traditional software artifacts are incompetent.

One such software artifact is contexts. They are structural information collected from the application's runtime environment to guide how it reacts to instant and complicated environmental events. Some examples of contexts are vision and voice instructions

to a humanoid companion robot[46], air composition at a specific site[47], and locations and speeds of vehicles[1,2]. It is vital for these applications to provide correct and real-time reactions. Otherwise, the users may suffer from economic or even personal losses.

This stirs up a strong demand for context consistency management with high efficiency. One way is data-centric checking, including noise inditification by assertions[48,49,50,51,52], anomaly filtering[48], fuzzy matching[53], sequence-based rules[49], watermarks[54], and probabilistic methods[55]. However, these approaches may miss many inconsistencies when they involve multiple context elements and complex relations between them.

Another way is constraint checking. It identifies and reasons about context inconsistencies by validating them against predefined rules[9,17,56,57]. These rules are about expected relations that the context elements should satisfy so they can achieve higher preciseness and soundness. This enables proper and timely resolution of context inconsistencies[19,20,21]. Researchers propose various techniques to accelerate constraint checking, e.g., full checking (ECC)[9], incremental checking (PCC)[11], CPU-based parallel checking (Con-C)[23], and GPU-based parallel checking (GAIN)[24]. Another interesting direction of efficiency optimization on constraint checking is selectively deciding the time points at which the constraint checking should be scheduled and suppressing unnecessary scheduling[25,12]. In this way, the times of scheduled checking turns are reduced, and the efficiency of constraint checking is indirectly improved.

6.2 | The Oracle Problem

Constraint checking is related to an expansive research topic about how to identify abnormal behaviors of software. It is formulated as the *oracle problem*[34]. The oracle problem focuses on how we could identify bugs when we could not derive expected outputs for complex software systems, e.g., compilers[58,59,60,61], language virtual machines[62,63], neural networks[64,65], and SMT solvers[66].

One way is differential testing, which tests the correctness of an implementation by cross-validating its outputs to other implementations[34,67,62]. However, the availability of multiple implementations[34] and the non-determinism in the system under test[67] may hinder the adoption of differential testing.

The second way is metamorphic testing that intentionally constructs multiple inputs, so the corresponding outputs must satisfy some specific relation[33]. The key point of metamorphic testing is finding a proper relation to which the inputs and outputs should conform. Such a relation is a semantic property of the software under testing that requires highly professional domain knowledge to discover. Some examples are behaviors of the compiled programs after dead code elimination[59], logical consistency of multiple logical formulas[66], and semantic consistency of image labels or query answers[64,65]. However, the difficulty of finding such good relations hinders metamorphic testing from being widely used.

Another way is assertions over each output[34]. Unlike in metamorphic testing, assertions check consistency within each single output. Similar to metamorphic testing, it is hard to write proper assertions that can depict the expected behavior[34]. Also, since assertions cannot express complex relations between multiple inputs and outputs, their code coverage may be limited[34].

Constraint checking can be used to conduct metamorphic testing or assertion testing. However, they have different focuses. Testing techniques like metamorphic testing and assertion testing care about the preciseness and completeness of found bugs, but constraint checking cares about the efficiency of this process.

6.3 | Constraint Solving

Constraint checking also relates to a more general field of constraint solving, where the properties of cyber-objects like programs can be expressed as logical constraints, and the satisfiability of these constraints relates the concerned objects to these properties. Such constraint solving techniques include popular Z3[68] and CVC4[69]. Vast applications can query constraint solvers to get information that assists their functionalities, e.g., symbolic execution tools[70,71] use them to decide whether a program path is feasible, program synthesizers[72] use them to compute possible solutions for integers in the program yet to synthesize, and program verifiers[73,74] leverage them to decide whether a program conforms to certain correctness specifications. The constraints checked by such constraint solvers are usually quite generic to cover as many application scenarios as possible and also simple enough to facilitate automatic collection. Due to these features of the constraints, constraint solvers confront various challenges such as the checking efficiency problem[75] and opaque third-party functions[76]. The context consistency management problem studied in this article usually requires high efficiency and customized manipulation of rich semantic information. Though with many works to improve the drawbacks of constraint solvers[77,78,78], their performance is still far from being able to fit in the constraint checking scenarios.

6.4 | Redundant Computation Reduction

Our work opens a potential new direction to support more efficient context inconsistency detection by removing redundant link generation and truth value evaluation rather than directly increasing the checking efficiency. It echoes redundant-computation-reduction work from other fields, e.g., avoiding redundant table scans to speed up the SQL-MapReduce task translation [79], avoiding redundant computations to speed up GNN training [80], and simplifying floating-point computations to speed up look-table operations in neural networks [81]. Although the principles are similar, these research efforts are closely bound to their subjects and thus not applicable to our problem.

7 | CONCLUSION

In this article, we address the issue of link redundancy in constraint checking and introduce a groundbreaking method, MG, designed to automatically detect and eliminate unnecessary link creation without affecting the outcome of checks. We have theoretically validated the soundness and completeness of MG and demonstrated its effectiveness through a comparative study with synthesized consistency constraints and a case study using extensive real-world context data. The findings confirm MG's capability to remove all link redundancy, enhancing link generation efficiency by 15–504x compared to previous methods. Furthermore, MG contributes to the detection of context inconsistencies by offering an additional efficiency increase, achieving up to a 45.4% reduction in time, which can be applied across all existing constraint checking techniques. By incorporating an escape-condition analysis that minimizes needless truth value evaluations, based on MG's thorough analysis, the enhanced version, MG+, achieves further reductions in constraint checking time by up to 28.2% beyond MG's improvements, cumulatively cutting the total checking duration by as much as 61.0%.

For our future research, we aim to explore strategies for automatically reformulating consistency constraints to eliminate redundancy inherently, moving beyond runtime solutions. This approach has the potential to be more cost-efficient for certain applications by ensuring that constraints are designed to be redundancy-free from the outset. Additionally, we plan to extend the validation of MG to encompass more complex constraints and dynamic application contexts, such as those involving unmanned drones and self-driving vehicles. This expansion may allow us to assess MG's applicability and effectiveness across a wider range of scenarios, thereby further broadening its utility.

ACKNOWLEDGMENTS

This work was supported by the Natural Science Foundation of China under Grant Nos. 61932021, 62302209, and 62072225, and the Leading-edge Technology Program of Jiangsu Natural Science Foundation under Grant Nos. BK20202001 and BK20220771. The authors would also like to thank the support from the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

References

1. Califronia DMV. 2020. autonomous vehicle disengagement reports. https://www.dmv.ca.gov/portal/file/2020-autonomous-vehicle-disengagement-reports-csv/; 2020. Accessed on Feb 8, 2022.

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

3. Ke C, Xiao F, Huang Z, Xiao F. A user requirements-oriented privacy policy self-adaption scheme in cloud computing. *Frontiers of Computer Science* 2023; 17(2): 172203.

4. Khoshnevis S. A search-based identification of variable microservices for enterprise SaaS. *Frontiers of Computer Science* 2023; 17(3): 173208. doi: 10.1007/s11704-022-1390-4

5. Harter A, Hopper A, Steggles P, Ward A, Webster P. The anatomy of a context-aware application. *Wireless Networks* 2002; 8(2): 187–197.

6. Cheng T, Zhao K, Sun S, Mateen M, Wen J. Effort-aware cross-project just-in-time defect prediction framework for mobile apps. *Frontiers of Computer Science* 2022; 16(6): 166207. doi: 10.1007/s11704-021-1013-5

7. Zhong Y, Shi M, Xu Y, Fang C, Chen Z. Iterative Android automated testing. *Frontiers of Computer Science* 2023; 17(5): 175212. doi: 10.1007/s11704-022-1658-8

8. Van Bunningen AH, Feng L, Apers PM. Context for ubiquitous data management. In: *International Workshop on Ubiquitous Data Management*; 2005: 17–24.

9. Nentwich C, Capra L, Emmerich W, Finkelsteiin A. xlinkit: A consistency checking and smart link generation service. *ACM Transactions on Internet Technology* 2002; 2(2): 151–185.

10. Xu C, Cheung SC, Chan WK. Incremental consistency checking for pervasive context. In: *Proceedings of the 28th International Conference on Software Engineering*; 2006: 292–301.

11. Xu C, Cheung SC, Chan WK, Ye C. Partial constraint checking for context consistency in pervasive computing. *ACM Transactions on Software Engineering and Methodology* 2010; 19(3): 1–61.

12. Xu C, Xi W, Cheung SC, Ma X, Cao C, Lu J. CINA: Suppressing the detection of unstable context inconsistency. *IEEE Transactions on Software Engineering* 2015; 41(9): 842–865.

13. Mao Z, Gu Y, Jiang B, Xu D, Sun X, Liu W. Incipient fault diagnosis for high-speed train traction systems via improved LSTM. *Scientia Sinica Informationis* 2021; 51(6): 997–1012. Original document in Chinese.

14. Poon PL, Lau MF, Yu YT, Tang SF. Spreadsheet quality assurance: a literature review. *Frontiers of Computer Science* 2024; 18(2): 182203.

15. Ranganathan A, Campbell RH. An infrastructure for context-awareness based on first order logic. *Personal and Ubiquitous Computing* 2003; 7(6): 353–364.

16. Park I, Lee D, Hyun SJ. A dynamic context-conflict management scheme for group-aware ubiquitous computing environments. In: *the 29th Annual International Computer Software and Applications Conference*; 2005: 359–364.

17. Bu Y, Gu T, Tao X, Li J, Chen S, Lu J. Managing quality of context in pervasive computing. In: *Proceedings of the Sixth International Conference on Quality Software*; 2006: 193–200.

18. Chomicki J, Lobo J, Naqvi S. Conflict resolution using logic programming. *IEEE Transactions on Knowledge and Data Engineering* 2003; 15(1): 244–249.

19. Xu C, Ma X, Cao C, Lu J. Minimizing the side effect of context inconsistency resolution for ubiquitous computing. In: *International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*; 2011: 285–297.

20. Xu C, Cheung SC, Chan WK, Ye C. 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: 569–572.

21. Xu C, Cheung SC, Chan WK, Ye C. Heuristics-based strategies for resolving context inconsistencies in pervasive computing applications. In: *2008 The 28th International Conference on Distributed Computing Systems*; 2008: 713–721.

22. Chen J, Qin Y, Wang H, Xu C. Simulation might change your results: a comparison of context-aware system input validation in simulated and physical environments. *Journal of Computer Science and Technology* 2022; 37(1): 83–105.

23. Xu C, Liu YP, Cheung SC, Cao C, Lu J. Towards context consistency by concurrent checking for internetware applications. *Science China Information Sciences* 2013; 56(8): 1–20.

24. Sui J, Xu C, Xi W, et al. GAIN: GPU-based constraint checking for context consistency. In: *Proceedings of the 21st Asia-Pacific Software Engineering Conference*; 2014; Jeju, South Korea: 319–326.

25. Wang H, Xu C, Guo B, Ma X, Lu J. Generic adaptive scheduling for efficient context inconsistency detection. *IEEE Transactions on Software Engineering* 2021; 47(03): 464-497. doi: 10.1109/TSE.2019.2898976

26. Zhang L, Wang H, Xu C, Yu P. INFUSE: Towards efficient context consistency by incremental-concurrent check fusion. In: *Proceedings of the 38th International Conference on Software Maintenance and Evolution*; 2022; Limassol, Cyprus. forthcoming.

27. Xu C, Qin Y, Yu P, Cao C, Lu J. Techniques for growing software: paradigm and beyond. *Scientia Sinica Informationis* 2020; 50(11): 1595–1611. Original document in Chinese.

28. Chen C, Wang H, Zhang L, Xu C, Yu P. Minimizing link generation in constraint checking for context inconsistency detection. In: *Proceedings of the 38th IEEE International Symposium on Software Relibability Engineering (ISSRE 2022)*; 2022; Charlotte, North Carolina, USA: 13–24.

29. Xu C, Cheung SC, Chan WK. Goal-directed context validation for adaptive ubiquitous systems. In: *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*; 2007; Minneapolis, Minnesota, USA: 1–10

30. Xu C, Cheung SC, Lo C, Leung KC, Wei J. Cabot: On the ontology for the middleware support of context-aware pervasive applications. In: *IFIP International Conference on Network and Parallel Computing*; 2004: 568–575.

31. Sui J, Xu C, Cheung SC, et al. Hybrid CPU–GPU constraint checking: towards efficient context consistency. *Information and Software Technology* 2016; 74: 230–242.

32. Chen C. The implementation of MG and MG+. https://github.com/cychen2021/mg/; 2024. online.

33. Chen TY, Cheung SC, Yiu SM. Metamorphic testing: a new approach for generating next test cases. 2020

34. Patel K, Hierons RM. A mapping study on testing non-testable systems. *Software Quality Journal* 2018; 26(4): 1373–1413. doi: 10.1007/s11219-017-9392-4

35. Gao M, Wang H, Xu C. Testing constraint checking implementations via principled metamorphic transformations. In: *Proceedings of the 31st IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2024)*; 2024; Rovaniemi, Finland. forthcoming.

36. Reiss SP. Incremental maintenance of software artifacts. *IEEE Transactions on Software Engineering* 2006; 32(9): 682–697.

37. Nentwich C, Emmerich W, Finkelsteiin A, Ellmer E. Flexible consistency checking. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2003; 12(1): 28–63.

38. Ma J, Sun QW, Xu C, Tao XP. Griddroid–an effective and efficient approach for android repackaging detection based on runtime graphical user interface. *Journal of Computer Science and Technology* 2022; 37(1): 147–181.

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

40. Blanc X, Mounier I, Mougenot A, Mens T. Detecting model inconsistency through operation-based model construction. In: *Proceedings of the 30th ACM/IEEE International Conference on Software Engineering*; 2008: 511–520.

41. ArgoUML. https://github.com/argouml-tigris-org/argouml; 2022. Accessed on Mar 6, 2022.

42. Demsky B, Rinard MC. Goal-directed reasoning for specification-based data structure repair. *IEEE Transactions on Software Engineering* 2006; 32(12): 931–951.

43. Chen C, Ye C, Jacobsen HA. Hybrid context inconsistency resolution for context-aware services. In: *2011 IEEE International Conference on Pervasive Computing and Communications*; 2011: 10–19.

44. Zhao Z, Huang D, Ma X. TOAST: Automated testing of object transformers in dynamic software updates. *Journal of Computer Science and Technology* 2022; 37(1): 50–66.

45. Demuth A, Riedl-Ehrenleitner M, Egyed A. Efficient detection of inconsistencies in a multi-developer engineering environment. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*; 2016: 590–601.

46. Kuo PH, Lin ST, Hu J, Huang CJ. Multi-sensor context-aware based chatbot model: an application of humanoid companion robot. *Sensors* 2021; 21(15): 5132.

47. Pollen Wise - what's in your air, when and where. https://play.google.com/store/apps/details?id=com.PollenSense.PollenWise; 2022. Accessed on May 19, 2022.

48. Jeffery SR, Garofalakis M, Franklin MJ. Adaptive cleaning for RFID data streams. In: *Proceedings of the 32nd International Conference on Very large Data Bases*; 2006: 163–174.

49. Rao J, Doraiswamy S, Thakkar H, Colby LS. A deferred cleansing method for RFID data analytics. In: *Proceedings of the 32nd International Conference on Very Large Data Bases*; 2006: 175–186.

50. Patil KT, Bansal V, Dhateria V, Narayankhedkar SK. Probable causes of RFID tag read unreliability in supermarkets and proposed solutions. In: *2015 International Conference on Information Processing*; 2015: 392–397.

51. Fescioglu-Unver N, Choi SH, Sheen D, Kumara S. RFID in production and service systems: Technology, applications and issues. *Information Systems Frontiers* 2015; 17(6): 1369–1380.

52. Want R. RFID: A key to automating everything. *Scientific American* 2004; 290(1): 56–65.

53. Chaudhuri S, Ganjam K, Ganti V, Motwani R. Robust and efficient fuzzy match for online data cleaning. In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*; 2003: 313–324.

54. Song Y, Li Y, Yang H, Xu J, Luan Z, Li W. Adaptive watermark generation mechanism based on time series prediction for stream processing. *Frontiers of Computer Science* 2021; 15(6): 156213.

55. Khoussainova N, Balazinska M, Suciu D. 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: 43–50.

56. Bu Y, Chen S, Li J, Tao X, Lu J. Context consistency management using ontology based model. In: *International Conference on Extending Database Technology*; 2006: 741–755.

57. Xu C, Cheung SC. Inconsistency detection and resolution for context-aware middleware support. *ACM SIGSOFT Software Engineering Notes* 2005; 30(5): 336–345.

58. Yang X, Chen Y, Eide E, Regehr J. Finding and understanding bugs in C compilers. In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*; 2011; New York, NY, USA: 283–294

59. Le V, Afshari M, Su Z. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices* 2014; 49(6): 216–226.

60. Chaliasos S, Sotiropoulos T, Spinellis D, Gervais A, Livshits B, Mitropoulos D. Finding typing compiler bugs. In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*; 2022; New York, NY, USA: 183–198

61. Liu J, Lin J, Ruffy F, et al. Finding deep-learning compilation bugs with NNSmith. 2022

62. Chen Y, Su T, Su Z. Deep differential testing of JVM implementations. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*; 2019: 1257–1268.

63. Li C, Jiang Y, Xu C, Su Z. Validating JIT compilers via compilation space exploration. In: *Proceedings of the 29th ACM Symposium on Operating Systems Principles*; 2023.

64. Xie X, Ho JW, Murphy C, Kaiser G, Xu B, Chen TY. Testing and validating machine learning classifiers by metamorphic testing. *Journal of Systems and Software* 2011; 84(4): 544–558.

65. Chen S, Jin S, Xie X. Testing your question answering software via asking recursively. In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*; 2021: 104–116

66. Winterer D, Zhang C, Su Z. Validating SMT solvers via semantic fusion. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*; 2020; New York, NY, USA: 718–730

67. Gulzar MA, Zhu Y, Han X. Perception and practices of differential testing. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*; 2019: 71-80

68. De Moura L, Bjørner N. Z3: An efficient SMT solver. In: *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, Held as Part of the Joint European Conferences on Theory and Practice of Software, Proceedings*; 2008; Budapest, Hungary: 337–340.

69. Barrett C, Conway CL, Deters M, et al. cvc4. In: *Computer Aided Verification: 23rd International Conference, Proceedings*; 2011; Snowbird, USA: 171–177.

70. Cadar C, Dunbar D, Engler DR, others . KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs.. In: *Proceedings of the 8th USENIX conference on Operating systems design and implementationDecember*; 2008: 209–224.

71. Godefroid P, Klarlund N, Sen K. DART: Directed automated random testing. In: *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*; 2005: 213–223.

72. Solar-Lezama A. *Program synthesis by sketching*. University of California, Berkeley . 2008.

73. DeLine R, Leino R. BoogiePL: A typed procedural language for checking object-oriented programs. tech. rep., Citeseer; 2005.

74. Detlefs D, Nelson G, Saxe JB. Simplify: A theorem prover for program checking. *Journal of the ACM* 2005; 52(3): 365–473.

75. Xu H, Zhao Z, Zhou Y, Lyu MR. Benchmarking the capability of symbolic execution tools with logic bombs. *IEEE Transactions on Dependable and Secure Computing* 2018; 17(6): 1243–1256.

76. Muduli SK, Roy S. Satisfiability modulo fuzzing: a synergistic combination of SMT solving and fuzzing. *Proceedings of the ACM on Programming Languages* 2022; 6(OOPSLA2): 1236–1263.

77. Pandey A, Kotcharlakota PRG, Roy S. Deferred concretization in symbolic execution via fuzzing. In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*; 2019: 228–238.

78. Mikek B, Zhang Q. Speeding up SMT solving via compiler optimization. In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*; 2023; New York, NY, USA: 1177–1189

79. Lee R, Luo T, Huai Y, Wang F, He Y, Zhang X. YSmart: Yet another SQL-to-MapReduce translator. In: *Proceedings of the 31st International Conference on Distributed Computing Systems*; 2011: 25–36

80. Jia Z, Lin S, Ying R, You J, Leskovec J, Aiken A. Redundancy-free computation for graph neural networks. In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*; 2020; New York, NY, USA: 997–1005

81. Razlighi MS, Imani M, Koushanfar F, Rosing T. LookNN: Neural network with no multiplication. In: *Design, Automation and Test in Europe Conference and Exhibition, 2007*; 2017: 1775–1780