Compositional Constructive Metamorphism for Testing Constraint Checking Implementations

Wenze Jin[†‡], Huiyan Wang[*†§], and Chang Xu[*†‡]
[†]State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
[‡]School of Computer Science, Nanjing University, Nanjing, China
[§]Software Institute, Nanjing University, Nanjing, China
wenzejin@smail.nju.edu.cn, {why, changxu}@nju.edu.cn

***Abstract*—Modern software can interact with dynamic environments, which span human, cyber, and physical elements. These elements form streaming context data, posing substantial challenges for correct and efficient handling. One approach is to deploy constraint checking (CC) to guard context consistency, with techniques focusing on checking performance or false warning mitigation. However, the correctness of CC implementations remains largely untouched due to the lack of test oracles to validate results. While metamorphic testing (MT) can partially alleviate this problem, existing work still relies on manually crafted metamorphic relations, which nevertheless restrict the structural diversity of test cases and lead to inadequate test coverage of CC implementations. In this paper, we propose Compositional Constructive Metamorphism (CCM), which reconceptualizes MT as a process of deconstructing and then reconstructing existing test efforts in a metamorphic manner, with diversity enforced and soundness proved. We evaluated CCM on three real-world CC implementations and 1,295 mutants. The results show that CCM clearly outperformed existing work, increasing the mutant killing rate by 17.9% (absolute) while achieving higher efficiency (requiring 84.7% fewer test cases and taking 26.6% less time). Furthermore, CCM detected three previously unknown bugs in subjects that had been intensively tested in prior work.**

***Index Terms*—context management, metamorphic testing, metamorphic relations.**

I. Introduction

Nowadays *constraint checking* (CC) techniques are being widely used for ensuring runtime reliability of modern software, e.g., self-adaptive systems [1], human-cyber-physical systems (HCPS) [2], [3]. These systems typically handle sensory data perceived from their environments (referred to as *contexts*), and then make adaptive decisions based on them. However, contexts can be error-prone due to unexpected and uncontrollable environmental noise [4], [5]. To address this, various CC techniques [6]–[11] have been proposed to improve context quality, e.g., by detecting and resolving context inconsistency, thereby enhancing system reliability. For example, we consider a cluster of UAV drones that continuously collect each drone's 3D coordinates for task planning. Since raw location sensory data are prone to random drift, CC techniques can serve as a validation middleware to guard data rationality, e.g., by checking, via some temporal *consistency constraints*, whether any calculated velocity between consecutive location reports remains within physical flying limits. When any inconsistency is encountered, the system can initiate proper remedial actions to recover locations from flying exceptions.

Most constraint checking techniques (e.g., PCC [6], ConC [7], GAIN [8], GEAS [9], MG [10], and INFUSE [11]) and their implementations (i.e., *CC implementations*) share a common interface, although they may have various sophisticated internal mechanisms. Typically, a CC implementation takes a consistency constraint and the contexts being checked as its inputs, and produces a corresponding checking result, consisting of a Boolean *truth value* indicating whether the constraint is satisfied or violated, along with a set of *links* that provide explanations for such satisfaction or violation. However, ensuring the correctness of these implementations is challenging due to the complex logic involved and the lack of a precise test oracle. For our aforementioned UAV scenario, regarding an arbitrary consistency constraint to check and huge-volume contexts with massive flight logs, it is practically infeasible to determine what the correct checking result should be. Thus prior research either relies on manually labeled datasets or conducts differential testing (for cross-validation) to detect possible CC implementation defects. Yet, these workarounds severely limit test automation and reliability, due to high labeling costs or functional discrepancies among CC implementations.

The only automation effort, Gao et al. [12], leverages the idea of *metamorphic testing (MT)*, which has proved successful in a broad range of applications that also suffer from the oracle problem, such as SMT solvers [13], compilers [14], [15], and large language models (LLMs) [16], [17]. Specifically, Gao et al. designed a new family of metamorphic relations (MRs) tailored for CC implementations, each consisting of an input transformation and an assertion. For each MR, their approach applies the transformation to the first input (a.k.a. *source input*, which can be an original constraint with contexts), and obtains the second input (a.k.a. *follow-up input*, which is a modified constraint with contexts resulting from this transformation). Then, the approach checks whether the assertion is satisfied, based on the two inputs' corresponding outputs from the CC implementations. Despite this approach's success, it also suffers from structural limitations and rigid patterns in its manually crafted MRs, which lead to inadequate and inefficient test coverage of CC implementations due to incomplete follow-up constraint structures. For example, these MRs are theoretically infeasible to transform a source constraint into a follow-up

* Corresponding authors.

one with a substantially different structure, e.g., from one rooted in a *binary connective* (like or) to one rooted in a *quantifier* (like $\forall$). This causes certain constraint structures to remain untouchable in some cases. Even for theoretically touchable constraint structures, some of them have an extremely low probability of triggering possible defects due to overly fixed transformation patterns. Such inadequacies can lead to incomplete fault detection capabilities, as our later evaluation shows.

To address these limitations, we propose a new shift in the traditional metamorphic testing paradigm by moving beyond the fixed catalog of manually crafted MRs. We conceptualize metamorphic testing for CC implementations as a generative process of deconstruction and subsequent reconstruction of existing test cases, and propose *Compositional Constructive Metamorphism (CCM)*. Unlike a traditional approach that treats source test cases as templates for transformation and relies on pre-specified MRs with a fixed structural modification space, CCM extracts *primitive units (PUs)*, i.e., building blocks enriched with inferred runtime semantics, from source test cases. Then, CCM maintains all collected primitive units in a central repository and actively selects them to systematically compose high-level complex PUs as follow-up test cases with predictable assertions. This enables a vastly expanded compositional space and yields new constraint structures of greater diversity and complexity. More importantly, each new constraint structure is associated with a valid, semantically grounded assertion, thereby enabling the automated derivation of diverse and effective metamorphic relations. In addition, we have formally proved that all test cases produced by CCM are guaranteed to pass for all correct CC implementations, ensuring its theoretical soundness.

We empirically evaluated CCM through comprehensive experiments involving three real-world CC implementations and their 1,295 mutants. The results demonstrate substantial gains in both test effectiveness and efficiency. CCM achieved a mutant killing rate of 92.7%, surpassing the existing approach by 17.9% (absolute). Notably, it also successfully revealed three previously unknown bugs in these CC implementations, while still covering all bugs detected by the existing approach. In terms of efficiency, CCM significantly reduced the testing overhead, achieving an 84.7% reduction in test case requirements, a 26.6% reduction in time to expose bugs for the same set of mutants, and costing only 48.0 seconds and 227 test cases on average to expose real-world bugs.

The remainder of this paper is organized as follows. Section II presents the background and motivation. Section III elaborates on our proposed CCM approach. Section IV evaluates its effectiveness and efficiency. Section V discusses the application of CCM. Section VI discusses related work, and Section VII concludes the paper.

II. Background & Motivation

A. Preliminary

A CC implementation is a program that checks the validity of the contexts collected by an application. To ease discussion, we follow existing work in modeling and assume that the program takes as input a *context pool*, which contains the application's contexts of interest, and a *consistency constraint*, which enforces the validity of those contexts. It then returns a *truth value* indicating whether the constraint is violated, along with a *link set* explaining the result.

The context pool (denoted as P) is typically defined as a set of contexts, i.e., $P = \{C_1, C_2, \dots\}$, where each context C_i refers to a piece of structured information about an application's running environment containing all interested elements, i.e., $C_i = \{e_1, e_2, \dots\}$. To ensure the quality of contexts and properly guide the application service, *consistency constraints* are typically formulated by a First-Order-Logic (FOL) language as in existing literature [6]–[11]:

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

Here, C denotes a specific context from the pool P, and v is a variable bound to an element $d \in C$. The terminal $bfunc$ represents a domain-specific Boolean function evaluating these bound variables to return a truth value (T or F). For any given context pool P and a consistency constraint S, the CC implementation (denoted as $\mathcal{I}_{CC}$) can evaluate P against S to output its evaluation result: $\mathcal{I}_{CC}(P, S) \rightarrow \langle T, L \rangle$.

The truth value part, $T \in \{\mathsf{T}, \mathsf{F}\}$, indicates whether the constraint is satisfied (T) or violated (F). It also outputs a link set explaining the result, $L = \{l_1, l_2, \dots\}$ where each link l contains multiple variable bindings: $l = \{v_1 \mapsto e_1, v_2 \mapsto e_2, \dots\}$, indicating the root variable assignments.

Take our UAV scenario as an example. As illustrated in the scenario panel of Fig. 1, the context pool contains two contexts that separately maintain the current status and the last status of all UAVs, $P_{\mathsf{UAV}} = \{C_{\mathsf{cur}}, C_{\mathsf{lst}}\}$. Consider the two UAVs (U_1 and U_2) in the scenario. The corresponding contexts C_{cur} and C_{lst} store their current and last location logs, including field information (i.e., timestamps, UAV identifier, location, and emergency status). Suppose that U_1 moved sharply from $\mathsf{loc_A}$ to $\mathsf{loc_C}$ at an unusual speed, whereas U_2 moved from $\mathsf{loc_B}$ to $\mathsf{loc_D}$ and suddenly invokes an emergency state as shown in Fig. 1. The current context pool includes $C_{\mathsf{cur}} = \{\langle \mathsf{t}_1, \mathsf{U}_1, \mathsf{loc_C}, \mathsf{F} \rangle(\mathsf{e}_1), \langle \mathsf{t}_1, \mathsf{U}_2, \mathsf{loc_D}, \mathsf{T} \rangle(\mathsf{e}_2)\}$ and $C_{\mathsf{lst}} = \{\langle \mathsf{t}_0, \mathsf{U}_1, \mathsf{loc_A}, \mathsf{F} \rangle(\mathsf{e}_3), \langle \mathsf{t}_0, \mathsf{U}_2, \mathsf{loc_B}, \mathsf{F} \rangle(\mathsf{e}_4)\}$.

In this scenario, a proper consistency constraint S_{UAV} can be formulated to describe the velocity rationality control for the entire cluster when no UAV in the cluster is in an emergency:

$$\begin{aligned} & \big(\forall x \in C_{\mathsf{cur}}\ \forall y \in C_{\mathsf{lst}} \big(\mathrm{same\_id}(x, y)\ \mathsf{implies}\ \mathrm{vel\_chk}(x, y)\big)\big) \\ & \quad \mathsf{or}\ \big(\exists z \in C_{\mathsf{cur}}\ \mathrm{e\_status}(z)\big). \end{aligned}$$

This constraint utilizes three domain-specific Boolean functions: $\mathrm{same\_id}$ and $\mathrm{vel\_chk}$ validate whether the velocity calculated from any UAV's consecutive flight statuses remains within reasonable limits, while $\mathrm{e\_status}$ identifies whether a UAV's emergency status is active (T). For simplicity, we denote this constraint as $S_{\text{UAV}} = (S_{\text{vc}})\ \mathsf{or}\ (S_{\text{emg}})$.

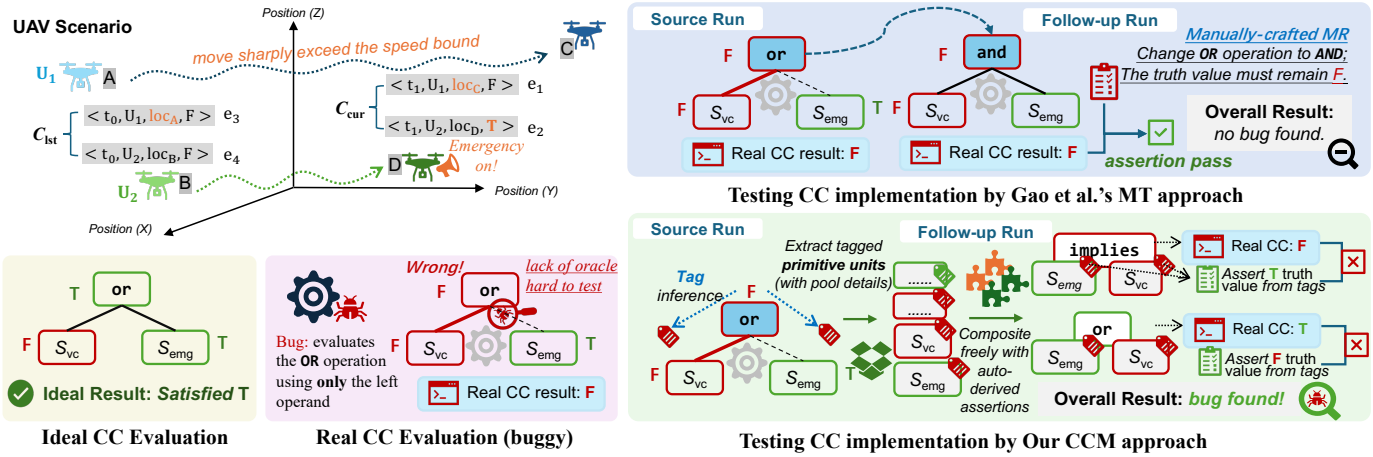

Fig. 1. An illustrative example based on the UAV scenario.

Now, to evaluate the constraint S_{UAV} based on the current context pool, the ideal result is satisfied since although the sharp, high-speed movement of U_1 violates the S_{vc} branch, the emergency state of U_2 satisfies the S_{emg} branch, thereby fulfilling the overall constraint. Therefore, an ideal $\mathcal{I}_{CC}$ returns T, and a link set with a link $l = \{z \mapsto \mathsf{e}_2\}$, indicating that binding variable z to e_2 makes the constraint satisfied. If we assume that U_2 is not in an emergency state currently, this would cause the whole constraint to be violated since now the velocity rationality is violated due to U_1's sharp movement. Then, in this case, an ideal $\mathcal{I}_{CC}$ returns F and a link set with a link $l = \{x \mapsto \mathsf{e}_1, y \mapsto \mathsf{e}_3\}$, indicating that binding x to e_1 and y to e_3 causes the violation.

B. The Oracle Problem and Metamorphic Testing

Testing practical CC implementations is challenging because it is intractable for testers to determine the exact expected output (T and L) for every possible input context pool against any possible consistency constraint to enforce on the artifacts. This barrier, recognized as the *oracle problem* [18], [19] in many fields, impedes effective testing of CC implementations. To address this, *Metamorphic Testing (MT)* [20] has emerged as a promising approach that, instead of specifying explicit oracles for programs under test, validates properties that must hold across multiple executions, known as metamorphic relations (MRs), and detects MR violations.

Inspired by this, Gao et al. [12] proposed the first automation approach for testing CC implementations with a family of manually crafted MRs. Operationally, Gao's approach begins with a *source execution* of the CC implementation under test, fed by (P_s, S_s) with a given context pool P_s and a well-formed consistency constraint S_s. Suppose the source execution results in a violation, and consider one of Gao's MRs: *if an* or*-rooted constraint is violated, changing its root to* and *must yield a constraint that is also violated against the same context pool*. As illustrated in the Gao's approach panel of Fig. 1, it would trigger a transformation process to generate a follow-up test case (P_f, S_f) where $P_f = P_s$ and $S_f = (S_{\text{vc}})$ and (S_{emg}), with an assertion A demanding the checking result $T_f = \mathsf{F}$. When the assertion condition is violated, the source and follow-up test cases together trigger and successfully expose a potential bug in the implementation.

C. Rethinking: From Mutation to Construction

While Gao's approach alleviates the oracle problem and shows positive effectiveness, its manually crafted MRs inherently limit the structural coverage of follow-up constraints, creating a false sense of reliability and flexibility.

Consider a buggy CC implementation with an "or-operation bug" that evaluates the or operation using only the left operand and ignoring the right, as shown at the Real CC Evaluation panel in Fig. 1. Given the aforementioned $S_{\text{UAV}} = (S_{\text{vc}})$ or (S_{emg}) and the context pool P_{UAV} where the U_2 drone has an active emergency status currently, the ideal CC evaluation should return T. However, the buggy implementation erroneously outputs F, as illustrated in the figure. In this case, due to the limited patterns in Gao's transformation (only changing the root or to and), the constraint becomes more restrictive. Consequently, the checking result in the follow-up execution remains F, thereby satisfying the assertion required by Gao's specified MR, and no bug is exposed. Furthermore, we exhaustively investigated all applicable MRs proposed by Gao et al. for this scenario and found them theoretically incapable of exposing this bug. This testing inadequacy highlights the bottleneck of Gao's mutative transformations, which are inherently restricted by the structure of the source constraint.

To address this limitation, CCM moves beyond the fixed catalog of manually crafted MRs and reconceptualizes metamorphic testing for CC implementations as a generative process of deconstructing and reconstructing existing test cases, shifting from mutative to *constructive* transformations. Taking the same hidden "or-operation bug" as an example, as shown in the CCM panel of Fig. 1, CCM extracts S_{UAV}'s sub-formulas into two primitive units, attaches P_{UAV} from the source execution to both, and infers their truth values from the source result to obtain each unit's corresponding tag. In this case, both tags are F for the S_{emg} and S_{vc} units, since the or root is evaluated to F in the source execution (despite the actual

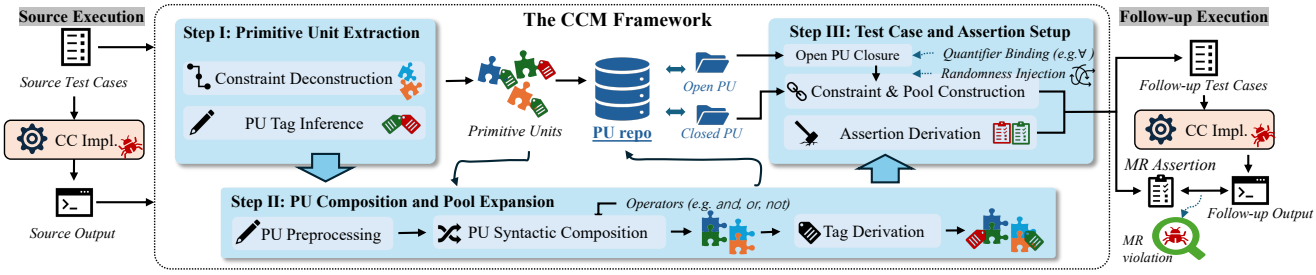

Fig. 2. The general workflow of CCM.

evaluation being T for S_{emg}). Each unit is enriched with a truth-value tag that captures a runtime property inferred from the implementation's final evaluation. These tagged units together populate a primitive pool, enabling CCM to systematically reconstruct diverse follow-up test cases with structural integrity and precise assertions.

After that, CCM composes possible primitive units in the pool freely with assertions, such as composing a follow-up constraint(S_{emg}) implies (S_{vc})with a T assertion derived from tagged values. By doing so, the follow-up execution leads to a constraint violation, since S_{emg} actually evaluates to T, thereby violating the assertion and successfully exposing the hidden bug. This constructive freedom and large compositional space significantly increase the likelihood of bug exposure, thereby improving testing adequacy. For instance, even when using or as the compositional operator (as illustrated in the figure), the same bug can also be successfully exposed.

III. Methodology

A. Overview

Fig. 2 presents an overview of CCM, which comprises three steps. First, **Primitive Unit Extraction** analyzes source test cases (constraints and context pools) and their outputs to extract primitive units (PUs) that encode semantic details of each constraint sub-formula. Second, **PU Composition and Pool Expansion** syntactically composes PUs with suitable semantic tags and adds the resulting PUs back to the repository, expanding both scale and complexity. Third, **Test Case and Assertion Setup** selects PUs to form testing inputs (e.g., binding free variables with quantifiers, injecting randomness into context pools) and derives assertions to verify follow-up executions as metamorphic properties; any assertion violation reveals a bug. We next define the core primitive units of CCM.

B. Primitive Units

A *primitive unit* (*PU*) is designed as a semantically grounded building block used to generate follow-up test cases that, together with the source test cases, serve as MR validation for testing CC implementations. Each PU indeed refers to a specific sub-constraint through limited but helpful semantic inference. It is defined as a triple: $u = \langle s, p, t \rangle$. Here, s represents a sub-constraint formula, typically extracted from a source constraint, p represents an associated context pool, and t represents a *semantic tag* encoding the semantic properties inferred from the source execution. We divide primitive units into *closed* and *open PUs* according to whether all variables in s are suitably bound by quantifiers (e.g. $\forall, \exists$).

A *closed PU* contains no free variables in its contained sub-constraint s, meaning s is structurally complete and can be directly evaluated by a CC implementation with its context pool p. Its semantic tag t records the expected evaluation result, including the truth value and the link result, i.e., $t = \langle T, L \rangle$. Consider a closed PU extracted from evaluating S_{UAV} in the UAV scenario: $u_1 = \langle S_{\text{emg}}, P_{\text{UAV}}, \langle T_1, L_1 \rangle \rangle$. Here, S_{emg} ($\exists z \in C_{\mathsf{cur}}(\text{e\_status}(z))$) contains no free variables since its only variable z is bound by quantifier $\exists$, i.e., its value is constrained by the context C_{cur}. This closed PU indeed indicates that evaluating S_{emg} on the context pool P_{UAV} should produce a result with truth value T_1 and link set L_1.

In contrast, an *open PU* contains free variables in s, making it structurally incomplete and unable to be evaluated directly. For example, if further deconstructing S_{vc} in S_{UAV}, one may obtain an open PU: $u_2 = \langle s_{\text{tmp}}, P_{\text{UAV}}, t_2 \rangle$, where $s_{\text{tmp}} = \forall y \in C_{\mathsf{lst}}(\text{same\_id}(x, y)\ \text{implies}\ \text{vel\_chk}(x, y))$, and x is not bound by any quantifier, thus a free variable. For this PU, its evaluation is non-deterministic, as the result of evaluating s_{tmp} upon pool P_{UAV} depends on the specific element binding of the free variable x. Therefore, to facilitate its evaluation expectations, the tag t must maintain these element-level details. Specifically, t for an open PU is required to record two disjoint contexts: C_{T} (including elements that satisfy s when bound to x) and C_{F} (including elements that violate s). Therefore, for an open unit, its tag is defined as $t = \langle v, C_{\mathsf{T}}, C_{\mathsf{F}} \rangle$, where v represents the free variable's name, while C_{T} and C_{F} specify concrete elements for v's contribution to the PU's associated formula satisfaction and violation, respectively. Suppose that during the source test case, one infers that binding x to element e_2 satisfies the s_{tmp} check while binding x to e_1 violates it, it can thus obtain the tag's requested contexts as $C_{\mathsf{T}} = \{\mathsf{e}_2\}$ and $C_{\mathsf{F}} = \{\mathsf{e}_1\}$. Accordingly, the extracted open PU is structured as $u_2 = \langle s_{\text{tmp}}, P_{\text{UAV}}, \langle x, \{\mathsf{e}_2\}, \{\mathsf{e}_1\} \rangle \rangle$. Maintaining these element-level details in PU tags enables CCM to later construct a suitable execution context for the open PU, by augmenting its existing context pool p with the recorded contexts C_{T} and C_{F} as candidate bindings for the free variable. Moreover, to control complexity, we restrict each open PU to contain at most one free variable.

C. Step I: Primitive Unit Extraction

In the first step, CCM extracts PUs from the source test cases and infers suitable semantic tags for them. Consider a source execution, where the CC implementation takes the source context pool P_{src} and constraint S_{src}, and outputs its checking result with truth value T_{src} and link set L_{src}, a.k.a., $\mathcal{I}_{CC}(P_{\text{src}}, S_{\text{src}}) \rightarrow \langle T_{\text{src}}, L_{\text{src}} \rangle$. Generally, CCM targets the source constraint S_{src} and extracts PUs in two ways: *decomposition-based* and *preservation-based*.

- In *decomposition-based extraction*, CCM directly deconstructs S_{src} into multiple sub-constraint formulas, each serving as a PU's constraint formula. These sub-constraints may be either closed or open, depending on whether they contain free variables. In this way, structural complexity is reduced by breaking the original constraint into smaller, more manageable fragments.
- In *preservation-based extraction*, CCM treats S_{src} itself (or a slight variation thereof) as a PU, optionally introducing free variables to maintain structural complexity. In this way, the structural complexity of the original constraint is preserved, enabling the generation of follow-up test cases that retain the original formula richness.

We now detail each PU extraction way, starting with the decomposition-based one, the main part for PU extraction.

Constraint Deconstruction. For a source constraint S_{src}, the decomposition first analyzes the root syntactic unit in S_{src}.

Connective Root. If S_{src} is rooted in a connective (e.g., $\mathsf{and}, \mathsf{or}, \mathsf{implies}$), the decomposition isolates its individual operands to produce sub-constraint formulas in PU. For each extracted sub-constraint s, the PU directly inherits the original context pool P_{src} as its associated context, without slicing. Since P_{src} is a superset that contains all bindings for the original constraint, it naturally subsumes any bindings that s may require. This ensures that each PU carries the complete contextual information, avoiding unnecessary complexity in pool management. Due to the natural closure of the original constraint S_{src}, each isolated sub-constraint s contains no free variables. Therefore, CCM can extract each s as a new closed PU with unknown tag t_{unk}: $u' = \langle s, P_{\text{src}}, t_{\text{unk}} \rangle$. For example, consider a source execution where $S_{\text{src}} = (S_{\text{vc}})\mathsf{or}(S_{\text{emg}})$ as aforementioned, with variables x, y taking elements from C_{cur} and C_{lst} respectively in S_{vc}, and z from C_{cur} in S_{emg}. Both sub-constraints under or yield closed PUs: $u' = \langle S_{\text{vc}}, P_{\text{src}}, t_{\text{unk}} \rangle$ and $u'' = \langle S_{\text{emg}}, P_{\text{src}}, t_{\text{unk}} \rangle$, where P_{src} is the original context pool (superset) containing C_{cur}, C_{lst}, and all relevant bindings.

Quantifier Root. If the source constraint is rooted in a quantifier (e.g., $Qx \in C(s)$ where $Q \in \{\forall, \exists\}$), the decomposition simply removes the quantifier to expose its inner sub-constraint s, which frees a previously bound variable x and results in an open PU. In this case, the PU also inherits the original context pool P_{src} as its associated context, since P_{src} (as a superset) already contains the context C for the quantifier. For example, consider a source constraint $S_{\text{src}} = \exists z \in C_{\text{cur}}(\text{e\_status}(z))$. Its extracted open PU is $u' = \langle \text{e\_status}(z), P_{\text{src}}, t_{\text{unk}} \rangle$, where P_{src} is also inherited directly as the original context pool (superset).

PU Tag Inference. For each extracted PU, we infer a semantic tag that denotes how its sub-constraint should be evaluated over the context pool. As defined in Section III-B, a closed PU's tag records the expected truth value and link set, while an open PU's tag specifies which bindings of its free variable satisfy or violate the constraint. Although tag inference and constraint deconstruction are performed simultaneously, we present them separately for clarity.

Closed PUs. For a closed PU extracted from a connective unit, its semantic tag records the expected truth value and link set, concerning its sub-constraint s's contributions to the global result. For ease of presentation, CCM introduces an attribution function, $\mathsf{Attr}(L_{\text{src}}, s)$, to parse the branch-level link set for only links relating to variables in s. For example, given $L_{\text{src}} = \{\{x \mapsto \mathsf{e}_1\}, \{x \mapsto \mathsf{e}_2, y \mapsto \mathsf{e}_3\}\}$ and a PU's s containing only x, $\mathsf{Attr}(L_{\text{src}}, s)$ retains only the x binding, yielding the attributed link set $L' = \{\{x \mapsto \mathsf{e}_1\}, \{x \mapsto \mathsf{e}_2\}\}$. With L', CCM resolves t_{unk} by jointly analyzing the connective's boolean logic. First, if the s's truth value T can be definitively deduced from the global truth value T_{src} (e.g., a globally satisfied and strictly requires both branches to be T), CCM directly assigns T and L' to form the tag: $u = \langle s, P_{\text{src}}, \langle T, L' \rangle \rangle$. Second, if direct boolean deduction is ambiguous (e.g., a globally satisfied or), CCM relies on L' as execution evidence. A non-empty L' ($L' \neq \emptyset$) proves s actively contributed to the global result, prompting CCM to assign the truth value T that justifies this contribution (e.g., $T = \mathsf{T}$ for an or connective) alongside L'. Conversely, an empty L' indicates an unverified contribution, leading to the conservative discarding of the incomplete PU. Consider the aforementioned example where $S_{\text{src}} = (S_{\text{vc}})\mathsf{or}(S_{\text{emg}})$ and the global result is $\langle \mathsf{T}, L_{\text{src}} \rangle$ with $L_{\text{src}} = \{\{z \mapsto \mathsf{e}_2\}\}$. Since the truth values of branches of a satisfied or are ambiguous, CCM analyzes L'. For S_{emg} (containing only z), $\mathsf{Attr}(L_{\text{src}}, S_{\text{emg}}) = \{\{z \mapsto \mathsf{e}_2\}\} \neq \emptyset$. This confirms its contribution, finalizing $u = \langle S_{\text{emg}}, P_{\text{src}}, \langle \mathsf{T}, \{\{z \mapsto \mathsf{e}_2\}\} \rangle \rangle$. Conversely, for S_{vc} (containing x, y), $\mathsf{Attr}(L_{\text{src}}, S_{\text{vc}}) = \emptyset$, whose incomplete PU is discarded without definitive tag evidence.

Open PUs. For an open PU extracted by removing a quantifier, its semantic tag must maintain the element-level details for its newly freed variable. CCM introduces $\mathsf{Extract}(L_{\text{src}}, x)$ for scanning the global link set L_{src} and collecting all elements that were bound to the free variable x within any link. This reveals the exact x's element bindings that triggered the global result. Depending on whether T_{src} is T or F, CCM assigns these extracted elements to either the satisfying context C_{T} or the violating context C_{F}, and populates the opposing context with the remaining elements from the context C of the original quantifier $Qx \in C(s)$. For example, consider $S_{\text{src}} = \exists z \in C_{\text{cur}}(\text{e\_status}(z))$. Suppose $C_{\text{cur}} = \{\mathsf{e}_1, \mathsf{e}_2, \mathsf{e}_3\}$ in P_{src}, and the global result is $\langle \mathsf{T}, L_{\text{src}} \rangle$ with $L_{\text{src}} = \{\{z \mapsto \mathsf{e}_1\}\}$. The extraction function pulls out the triggering element: $\mathsf{Extract}(L_{\text{src}}, z) = \{\mathsf{e}_1\}$. Because the overall result is T, this extracted element represents the satisfying one. Thus, $C_{\mathsf{T}} = \{\mathsf{e}_1\}$, while the remaining elements form the violating context $C_{\mathsf{F}} = \{\mathsf{e}_2, \mathsf{e}_3\}$. CCM then finalizes the open unit as $u = \langle \text{e\_status}(z), P_{\text{src}}, \langle z, \{\mathsf{e}_1\}, \{\mathsf{e}_2, \mathsf{e}_3\} \rangle \rangle$.

Despite the decomposition-based extraction, CCM also performs preservation-based extraction to maintain complexity, which is more straightforward and directly uses the source constraint S_{src} for a standalone PU. Since S_{src} is inherently structurally valid and has no free variables, CCM directly extracts it, along with P_{src}, as a closed PU. It also assigns the source evaluation result as the semantic tag: $u = \langle S_{\text{src}}, P_{\text{src}}, \langle T_{\text{src}}, L_{\text{src}} \rangle \rangle$. Furthermore, to harvest more open PUs with additional structural complexity, CCM conceptualizes S_{src} as an open formula by introducing an additional free variable m. To formalize this, we define a universal context C_{U} representing the set of all possible elements (i.e., for any context C_i, $C_{\mathsf{U}} \cup C_i = C_{\mathsf{U}}$). Because S_{src} does not actually contain m, its source truth value T_{src} holds regardless of how m is bound. Thus, if $T_{\text{src}} = \mathsf{T}$, all elements satisfy the constraint, yielding the open PU $u = \langle S_{\text{src}}, P_{\text{src}}, \langle m, C_{\mathsf{U}}, \emptyset \rangle \rangle$. Conversely, if $T_{\text{src}} = \mathsf{F}$, all elements violate it, yielding $u = \langle S_{\text{src}}, P_{\text{src}}, \langle m, \emptyset, C_{\mathsf{U}} \rangle \rangle$. All extracted PUs are deposited into a central PU repository.

D. Step II: PU Composition and Repository Expansion

With the central PU repository populated, CCM systematically constructs new, more complex PUs from existing ones and feeds the resulting PUs back into the repository to expand it. This generative process relies on three sequential sub-steps:

PU Preprocessing. First, CCM selects existing PUs from the repository. If the selected ones share identical variable or context names, combining them directly would compromise their independence and make subsequent tag deviations confusing and difficult to follow. Thus, CCM applies preprocessing to systematically rename variables and contexts across the PUs by assigning unique PU identifiers, ensuring absolute naming uniqueness. Since it is a general syntactic renaming, PUs' original semantics are still preserved.

PU Syntactic Composition. Following preprocessing, CCM applies syntactic composition to combine PUs. Specifically, CCM attaches either a unary operator (e.g., not) to a single original PU or binary connectives (e.g., and, or, implies) to connect two original PUs, forming a new structurally complex sub-constraint s_{new} based on the PUs' associated sub-constraints. For open PUs, we unify their free variables to a single fresh one if they are connected by a connective and merge their associated context pools to form a unified pool p_{new}. Notably, CCM intentionally defers the use of quantifiers to the final step, because introducing a quantifier binds a free variable and prematurely halts further composition for open PUs. In this way, CCM can result in each composed PU with its tag unknown, e.g., $u' = \langle s_{new}, p_{new}, t_{\text{unk}} \rangle$.

Tag Derivation. Then, CCM derives the semantic tag t_{new} for the newly composed PU based on its dependent PUs in composition according to our derivation logic, exhaustively detailed in Table I and Table II. PUs coming from attaching not are straightforward, which only need to reverse the original PU's truth value. Here, we explain the rules for attaching binary connectives when composing PUs in two categories.

Category 1: Composing Closed PUs. For each composed closed PU, its tag must contain the corresponding truth value and link set derived from the original PUs. Deriving the new truth value T_{new} can be straightforward, following standard Boolean logic for each binary connective. Deriving the new link set L_{new} requires applying specific set operations based on the connective's semantic requirements. In general, when multiple original PUs jointly contribute to the result of the composed PU, the resulting link set either (i) reflects only the link set of the specific branch that determines the result, or (ii) unifies the results contributed by both branches (combined via Cartesian product). For example, when combining two existing PUs for a newly composed PU using an and and now supposing the resulting truth value becomes T, the satisfaction result is equally determined by its two branches (two existing PUs). Therefore, CCM employs a Cartesian Product operation ($\otimes$) to multiply the source link sets for the resulting link set. For instance, consider combining two PUs whose tags are $\langle \mathsf{T}, \{\{x \mapsto \mathsf{e}_1\}, \{y \mapsto \mathsf{e}_2\}\} \rangle$ and $\langle \mathsf{T}, \{\{z \mapsto \mathsf{e}_3\}\} \rangle$. Composing them with and strictly pairs every valid binding, yielding $\langle \mathsf{T}, \{\{x \mapsto \mathsf{e}_1, z \mapsto \mathsf{e}_3\}, \{y \mapsto \mathsf{e}_2, z \mapsto \mathsf{e}_3\}\} \rangle$ via the Cartesian Product. Otherwise, if the resulting PU's truth value is alternatively determined by its branches, e.g., attaching an or connective for a composed PU with T result, CCM utilizes a Union operation ($\cup$) to merge link sets of the contributing PUs (when $T_i = \mathsf{T}$). Using the same PUs, composing them with or yielding $\langle \mathsf{T}, \{\{x \mapsto \mathsf{e}_1\}, \{y \mapsto \mathsf{e}_2\}, \{z \mapsto \mathsf{e}_3\}\} \rangle$ via the Union. Ultimately, the derived tag $\langle T_{new}, L_{new} \rangle$ is attached to form the new closed PU, i.e., $u' = \langle s_{new}, p_{new}, \langle T_{new}, L_{new} \rangle \rangle$.

Category 2: Composing Open PUs. The key step in composing open PUs is to unify the element bindings from dependent PUs for the free variables. First, to control the number of free variables, CCM renames each dependent PU's free variable to a fresh unified name z (i.e., the unified free variable in the derived tag). Then, CCM computes the new satisfying context C'_{T} and violating context C'_{F} using set operations. The specific set operations are determined by the semantics of the composition operator: intersection ($\cap$) represents concurrent requirements (e.g., both sub-constraints must be satisfied), while union ($\cup$) represents alternative requirements (e.g., at least one sub-constraint must be satisfied). For example, composing two open PUs with and yields $C'_{\mathsf{T}} = C_{\mathsf{T}1} \cap C_{\mathsf{T}2}$ (both must satisfy) and $C'_{\mathsf{F}} = C_{\mathsf{F}1} \cup C_{\mathsf{F}2}$ (violating either suffices). The resulting tag $\langle z, C'_{\mathsf{T}}, C'_{\mathsf{F}} \rangle$ completes the new open PU. These newly derived PUs are fed back into the central PU repository. While this generative cycle could theoretically be applied iteratively to increase complexity and diversity, we limit it to two rounds to control repository scale.

E. Step III: Test Case and Assertion Setup

In the final step, CCM randomly retrieves a PU from the repository and constructs a follow-up test case (P_f, S_f) with an assertion A to test the CC implementation. Since the PU repository can grow to millions of candidates, we employ an Adaptive Random Selection (ARS) strategy based on tree edit distance to maximize structural diversity. After an initial

TABLE I
OPERATIONAL RULES FOR COMPOSITION OF CLOSED PRIMITIVE UNITS

Operator	Composition Rule
Not	$u_{new} = \langle \mathsf{not}(s), p, \langle \neg T, L \rangle \rangle$
And	$u_{new} = \langle (s_1)\mathsf{and}(s_2), p_1 \cup p_2, \langle T', L' \rangle \rangle, \quad T' = T_1 \wedge T_2;$ $L' = \begin{cases} L_1 \otimes L_2 & \text{if } T' = \mathsf{T} \\ \bigcup \{L_i \mid T_i = \mathsf{F}\} & \text{if } T' = \mathsf{F} \end{cases}$
Or	$u_{new} = \langle (s_1)\mathsf{or}(s_2), p_1 \cup p_2, \langle T', L' \rangle \rangle, \quad T' = T_1 \vee T_2;$ $L' = \begin{cases} \bigcup \{L_i \mid T_i = \mathsf{T}\} & \text{if } T' = \mathsf{T} \\ L_1 \otimes L_2 & \text{if } T' = \mathsf{F} \end{cases}$
Implies	$u_{new} = \langle (s_1)\mathsf{implies}(s_2), p_1 \cup p_2, \langle T', L' \rangle \rangle$ Derived logically via structural equivalence: $(s_1)\mathsf{implies}(s_2) \equiv (\mathsf{not}(s_1))\mathsf{or}(s_2)$

Note: Unary $u = \langle s, p, t \rangle$; binary $u_i = \langle s_i, p_i, t_i \rangle$. For closed units, $t_i = \langle T_i, L_i \rangle$. $\wedge, \vee$ are standard Boolean operators.

TABLE II
OPERATIONAL RULES FOR COMPOSITION OF OPEN PRIMITIVE UNITS

Operator	Composition Rule
Not	$u_{new} = \langle \mathsf{not}(s), p, \langle x, C_{\mathsf{F}}, C_{\mathsf{T}} \rangle \rangle$
And	$u_{new} = \langle (s_1[z/x])\mathsf{and}(s_2[z/y]), p_1 \cup p_2, \langle z, C_{\mathsf{T}1} \cap C_{\mathsf{T}2}, C_{\mathsf{F}1} \cup C_{\mathsf{F}2} \rangle \rangle$
Or	$u_{new} = \langle (s_1[z/x])\mathsf{or}(s_2[z/y]), p_1 \cup p_2, \langle z, C_{\mathsf{T}1} \cup C_{\mathsf{T}2}, C_{\mathsf{F}1} \cap C_{\mathsf{F}2} \rangle \rangle$
Implies	$u_{new} = \langle (s_1[z/x])\mathsf{implies}(s_2[z/y]), p_1 \cup p_2, \langle z, C_{\mathsf{F}1} \cup C_{\mathsf{T}2}, C_{\mathsf{T}1} \cap C_{\mathsf{F}2} \rangle \rangle$

Note: Unary $u = \langle s, p, t \rangle$; binary $u_i = \langle s_i, p_i, t_i \rangle$. For open units, $t_i = \langle v_i, C_{\mathsf{T}i}, C_{\mathsf{F}i} \rangle$. z: fresh variable for unification. $s[z/x]$: syntactic substitution of x with z in s.

random selection, CCM samples batches of candidates that are structurally furthest from previously tested PUs. This cycle repeats until the testing budget is exhausted, after which the pipeline advances to the next batch of source test cases. For each PU, CCM constructs the test case as follows.

Constraint & Pool Construction. CCM first constructs a constraint under test and a context pool based on the selected PU. For a closed PU $u_{\mathsf{c}} = \langle s_{\mathsf{c}}, p_{\mathsf{c}}, \langle T, L \rangle \rangle$, CCM directly initializes the test case as $S_f = s_{\mathsf{c}}$ and $P_f = p_{\mathsf{c}}$. To enhance diversity, CCM applies a Randomness Injection operation to P_f. For every variable v_i appearing in the link set L, this operation injects random elements into its corresponding context C_i. Since this injection only adds new elements to contexts that already contain critical elements (i.e., those that contribute to the overall result), it does not interfere with the critical evaluations that determine the final result, and rather, it merely introduces potential new evaluations.

Open PU Closure. If the selected PU is now open, i.e., containing free variable x. For example, an open PU $u_{\mathsf{o}} = \langle s_{\mathsf{o}}, p_{\mathsf{o}}, \langle m, C_{\mathsf{T}}, C_{\mathsf{F}} \rangle \rangle$, CCM first performs open PU closure to systematically close the constraint by suitably binding a quantifier $Q \in \{\forall, \exists\}$ and creating a new target context C_{target} to bind the free variable m. This operation successfully yields a structurally complete follow-up constraint $S_f = Qm \in C_{\mathsf{target}}(s_{\mathsf{o}})$. After the constraint is closed, CCM initializes the pool by integrating the newly created context: $P_f = p_{\mathsf{o}} \cup \{C_{\mathsf{target}}\}$. To determine the exact selection of quantifier Q, the element composition of C_{target}, and whether to apply randomness injection, CCM employs a *Semantic Forcing* strategy based on two directional goals. To deliberately enforce satisfaction (T), CCM seeds C_{target} exclusively using elements from the satisfying context C_{T}. If Q is universal ($\forall$), CCM strictly ensures $C_{\mathsf{target}} \subseteq C_{\mathsf{T}}$ with absolutely no Randomness Injection allowed. If Q is existential ($\exists$), CCM ensures $C_{\mathsf{target}} \cap C_{\mathsf{T}} \neq \emptyset$ while freely applying Randomness Injection to add random elements as noise. Conversely, to trigger a constraint violation (F), CCM seeds C_{target} from C_{F}. For a universal constraint ($\forall$) to break, CCM ensures $C_{\mathsf{target}} \cap C_{\mathsf{F}} \neq \emptyset$ alongside randomness injection. For an existential constraint ($\exists$) to fail entirely, CCM strictly ensures $C_{\mathsf{target}} \subseteq C_{\mathsf{F}}$ with no randomness injection.

Assertion Derivation. Finally, after preparing the constraint and the context pool for checking, CCM derives assertion A that should hold when the follow-up execution is performed.

Closed PUs. For a closed PU, the derived assertion demands that the follow-up truth value strictly matches the expected tag ($T_f == T$). Furthermore, to accommodate the randomness injection, which may introduce new links or expand existing links since new evaluations are introduced, the assertion allows that the actual follow-up link set L_f must be an extension of the original link set L (denoted as $L_f \sqsupseteq L$). This extension relation asserts that for every original critical link $l \in L$, there exists a corresponding link l' in L_f that fully encapsulates those original bindings ($l \subseteq l'$).

Open PUs. For an open PU, the derived assertion directly maps to the employed semantic forcing strategy. To force satisfaction, if Q is universal ($\forall$), the assertion demands $A = (T_f == \mathsf{T} \wedge L_f == \emptyset)$ ($\forall$ only generates links when violated). Otherwise, if Q is existential ($\exists$), the assertion employs the previously defined $\mathsf{Extract}$ function to verify that the CC implementation correctly points out the seeded satisfying elements: $A = (T_f == \mathsf{T} \wedge \mathsf{Extract}(L_f, x) \supseteq (C_{\mathsf{target}} \cap C_{\mathsf{T}}))$. To force violation, the logic diametrically mirrors this. For a broken universal constraint ($\forall$), the assertion demands $A = (T_f == \mathsf{F} \wedge \mathsf{Extract}(L_f, x) \supseteq (C_{\mathsf{target}} \cap C_{\mathsf{F}}))$; for a failed existential constraint ($\exists$), it demands $A = (T_f == \mathsf{F} \wedge L_f == \emptyset)$ ($\exists$ only generates links when satisfied).

If any PU's assertion in a follow-up test case is violated, CCM considers it an MR violation, and the source and follow-up test cases together serve as bug-triggering inputs.

F. Theoretical Soundness

To formally ensure that CCM avoids false positives (i.e., erroneously reporting bugs in a correct implementation), we first define a primitive unit (PU) as *precise* if its semantic tag strictly meets the expectations defined in Section III-B. Based on this, we present our core soundness theorem:

Theorem (Soundness). *Any correct CC implementation satisfies all test cases and assertions generated by CCM.*

The proof sketch follows the three steps of our methodology: we show that (i) initial PUs extracted from a correct implementation are precise, (ii) composing precise PUs preserves precision, and (iii) test cases and assertions derived from a precise PU are always satisfied by a correct implementation. By chaining these arguments, the end-to-end reliability of CCM's

testing process is guaranteed. The complete formalizations and proofs are provided on our website [21].

IV. Evaluation

We evaluate CCM via a controlled experiment on mutants and a practical evaluation on real-world CC implementations.

A. Research Questions

- **RQ1 (Effectiveness)**: How effective is our CCM in detecting mutated bugs in CC implementations, as compared to existing work?
- **RQ2 (Efficiency)**: How efficient is our CCM in detecting mutated bugs in CC implementations?
- **RQ3 (Usefulness)**: To what extent can CCM discover practical bugs in real-world CC implementations?

B. Experimental Design and Setup

1) Subjects: To ensure a fair comparison across the controlled experiments in RQ1 and RQ2, we adopt the same mutant design and generation methodology as in existing work [12]. Specifically, we use INFUSE [22] (the latest and most comprehensive CC implementation) and, based on PIT [23], systematically inject artificial defects for mutated implementations. After filtering out equivalent or irrelevant mutants, we obtain a total of 1,295 CC implementation mutants.

To assess CCM's usefulness for real-world evaluation in RQ3, we select three real-world CC implementations: CMID [24], MG [25], and INFUSE [22], and construct a dataset comprising six distinct CC implementation versions. This includes four historical versions with documented real-world bugs curated by Gao et al. [12] (denoted as CMID_B, MG_B, $\text{INFUSE}_{\text{B1}}$, and $\text{INFUSE}_{\text{B2}}$), providing a ground truth for known bugs. Furthermore, we include the latest, currently deployed versions of MG and INFUSE (MG_L and INFUSE_L) to evaluate CCM's capability to discover unknown bugs.

2) Process: We establish a standardized pipeline in which each testing approach is sequentially applied to every target subject. To bootstrap test generation, we systematically enumerate valid input constraints. The evaluated approaches automatically pair these constraints with randomly generated context pools to construct the initial source test cases. Following Gao et al. [12], we bound the structural depth to three layers for the controlled experiment (to manage computational overhead) and to four layers for the real-world evaluation (to explore deeper program states). During execution, we enforce a strict *fail-fast* mechanism: an approach terminates immediately upon an assertion violation or crash, logging the runtime metrics. To prevent indefinite execution, we impose strict time and test case budgets, which also serve as independent variables for our evaluation. Finally, to account for the inherent randomness in test generation, all experiments are repeated three times, and we report the average results.

3) Setup: We design independent and dependent variables.

Independent Variables. We control three key factors:

- *Baselines.* We evaluate **CCM** against: **SOTA-Gao** (the state-of-the-art CC testing approach by Gao et al. [12]) and **Rand-Crash** (a random testing baseline serving as a sanity check, which detects only execution crashes).
- *Time Budget.* We control the execution time allocated per target at discrete intervals, e.g., 1, 2, 5, and 10 minutes.
- *Case Budget.* We control the maximum number of *oracle-equipped test cases* (i.e., executable cases containing explicit assertions) permitted to be generated per target (e.g., 500, 1000, 5000). For Rand-Crash, all generated inputs are counted against this budget.

Dependent Variables. We measure two outcomes:

- *Detection Effectiveness.* For the controlled experiment (RQ1), we use the *Mutation Score* (i.e., the ratio of killed mutants to total valid mutants). For the real-world evaluation (RQ3), we measure *Bug Revelation* (i.e., whether an approach successfully triggers a documented or unknown bug within the allocated budget).
- *Detection Cost.* We measure bug-revealing cost as (i) *Time2Bug*: wall-clock time to bug report; and (ii) *Case2Bug*: cumulative test cases up to the triggering case. For non-detections, both metrics are set to their maximum budget values to prevent statistical bias.

4) To Answer: To answer **RQ1** (Effectiveness), we evaluate CCM's bug detection capability using the *Mutation Score* across all discrete configurations of *Time Budget* and *Case Budget*, as compared to both SOTA-Gao and Rand-Crash approaches. To answer **RQ2** (Efficiency), we specifically focus on the subset of mutants successfully killed by both CCM and SOTA-Gao, and within this mutually killed subset, we perform a detailed case-by-case comparison of the precise cost metrics (*Time2Bug* and *Case2Bug*). Finally, to answer **RQ3** (Usefulness), we analyze the *Bug Revelation* rate and the associated cost metrics across the six real-world CC versions, comparing CCM against SOTA-Gao.

5) Configuration: All experiments were conducted on a server equipped with a 48-core Intel Xeon Gold 5118 CPU @ 2.00 GHz and 128GB RAM, running Ubuntu 22.04.1 LTS, Python 3.10.12, and OpenJDK 21.0.7.

C. Experimental Results

1) RQ1 (Effectiveness): We evaluate the bug-detection capabilities of CCM against SOTA-Gao and Rand-Crash approaches across varying time and case budgets. Fig. 3 illustrates the mutation scores for the three approaches.

From the figure, we observe that CCM consistently and significantly outperforms both SOTA-Gao and Rand-Crash across all configurations. For example, CCM achieved mutation scores of 82.0%–92.7% compared to SOTA-Gao's 60.7%–74.8% and Rand-Crash's 22.8%–23.9% when the time budget was controlled from one to ten minutes. Notably, CCM's strictest case-budget result (83.7% at 500 cases) still exceeds Gao's peak at twenty times the budget (74.8% at 10,000 cases), and its 1-minute result (82.0%) surpasses Gao's 10-minute result (74.8%).

With budget increases, CCM shows a stable improvement of 10.7% (time) and 6.5% (cases), while SOTA-Gao plateaus near 5,000 cases due to inability to generate further valid follow-up

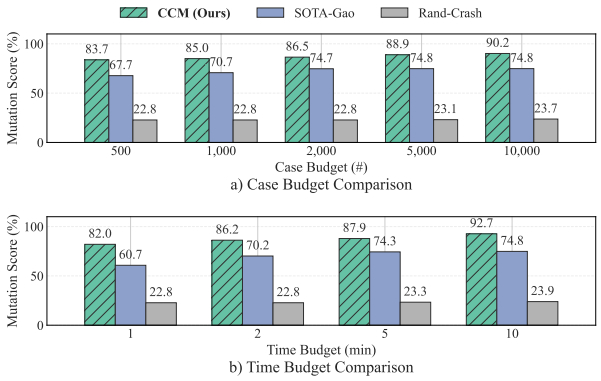

Fig. 3. Mutation scores achieved by CCM, SOTA-Gao, and Rand-Crash under discrete case budgets (a) and time budgets (b).

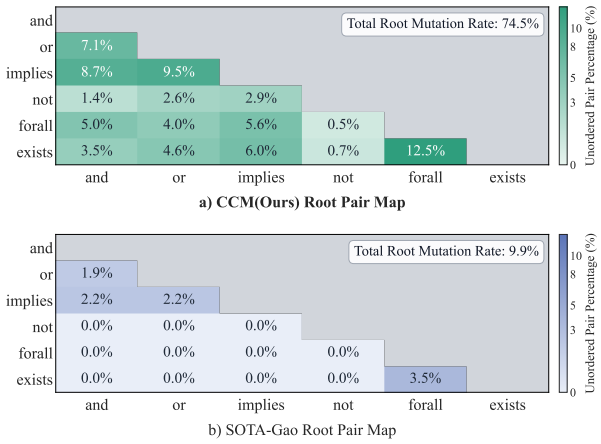

Fig. 4. Heatmap of cross-root transformation pairs between source and follow-up constraints for CCM and SOTA-Gao.

cases, and Rand-Crash improves only marginally (1.1% and 0.9%).

We further investigated the drivers of CCM's superior effectiveness by exploring structural diversity across source and follow-up constraints. We visualized unordered root-pair transformations (excluding identical pairs) to isolate effective root mutations, as shown in Fig. 4. The figure shows that CCM achieves a root mutation rate of 74.5% (covering all cross-root cells), while SOTA-Gao reaches only 9.9%, with most cross-root cells at 0.0%. This indicates that SOTA-Gao's limited patterns leave it trapped in preserving original structures. In contrast, CCM's heterogeneous transformation space enables it to alter constraint structures, potentially search deeper program states, and trigger complex bugs that structure-preserving baselines inherently miss.

Answer to RQ1: *CCM shows strong bug-detection effectiveness, achieving mutation scores of 82.0%–92.7% compared to SOTA-Gao's 60.7%–74.8%, with cross-budget dominance (e.g., its 1-minute results exceed Gao's 10-minute results). Its 74.5% root mutation rate (vs. Gao's 9.9%) suggests that structural diversity contributes to this superiority.*

2) RQ2 (Efficiency): To evaluate efficiency, we compare costs on identical bugs. We isolate 657 non-trivial mutants killed by both CCM and SOTA-Gao under their best configurations (excluding those trivial enough for Rand-Crash). Fig. 5 shows the time and case costs for this mutually killed subset.

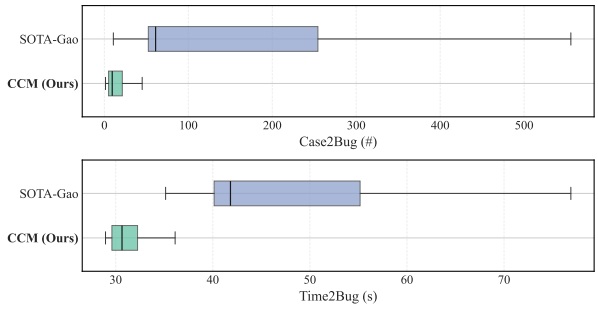

Fig. 5. Box plots (boxes indicate 25%–75% IQR; outliers excluded) showing the distribution of testing costs on the mutually killed mutants.

CCM requires a median of merely 9.3 test cases (Case2Bug) to trigger a bug, compared to Gao's 61.0 cases, constituting an **84.7% reduction**. This substantial reduction in generated cases directly explains CCM's reduction in time cost, as it achieves a median Time2Bug of 30.7s, a 26.6% reduction compared to SOTA-Gao's 41.8s. Moreover, CCM maintains an extremely compact IQR (29.6–32.3s) with negligible outliers, confirming its stable efficiency, whereas SOTA-Gao suffers from a severe execution tail exceeding 400s (outside the boxplot). The Wilcoxon signed-rank test [26] confirms that CCM's efficiency improvements are highly significant for both metrics ($p < 0.001$), with near-perfect rank-biserial correlation (0.99 for Time2Bug, 0.98 for Case2Bug).

We attribute this gain to the transformation diversity demonstrated in RQ1. By applying diverse constraint transformations, CCM generates test cases with higher semantic density. Consequently, instead of repeatedly probing localized program states, CCM rapidly traverses complex execution paths, triggering deep bugs with 84.7% fewer attempts. This algorithmic precision minimizes the Case2Bug metric, which in turn tightly bounds the overall Time2Bug.

Answer to RQ2: *CCM demonstrates exceptional efficiency. When detecting identical non-trivial bugs, CCM achieves an 84.7% reduction in the median number of test cases consumed (Case2Bug) and a 26.6% reduction in the median testing time (Time2Bug) compared to SOTA-Gao.*

3) RQ3 (Usefulness): We evaluate CCM and SOTA-Gao against six real-world versions of CC implementations under a generous 24-hour time budget and an unlimited case budget (Rand-Crash is not included because its lack of a test oracle makes it unlikely to expose any real-world bugs). Table III details the bug revelation results alongside their specific costs.

Overall, CCM achieves 100% bug revelation across all 5 documented bugs and additionally reports 3 previously unknown ones (Bug #3, #5, #8) in the latest deployed versions of CMID, MG, and INFUSE. On mutually detected bugs, CCM also exhibits vastly superior efficiency. For instance, concerning Bug #6, CCM requires merely 21.8s and 8 test cases, drastically outperforming SOTA-Gao's 630.9s and 13,231 cases. SOTA-Gao's bottleneck stems from substantial overhead in handling complex structural constraints for follow-up case generation.

TABLE III
BUG REVELATION RESULTS IN REAL-WORLD SUBJECTS

Version	Bug	CCM (Ours)			SOTA-Gao		
		Result	Time (s)	Cases (#)	Result	Time (s)	Cases (#)
$CMID_B$	#1	✓	**11.8**	**37**	✓	103.6	643
	#2	✓	**12.5**	**2**	✓	93.6	47
	#3*	✓	**11.9**	**3**	✗	–	–
MG_B	#4	✓	**30.5**	**25**	✓	387.2	56
MG_L	#5*	✓	**36.6**	**2**	✗	–	–
$INFUSE_{B1}$	#6	✓	**21.8**	**8**	✓	630.9	13,231
$INFUSE_{B2}$	#7	✓	**45.1**	63	✓	444.0	**49**
$INFUSE_L$	#8*	✓	**213.7**	**1,679**	✗	–	–

** indicates previously unknown bugs discovered by our approach.*

Even on Bug #7, where Gao uses slightly fewer cases (49 vs. 63), CCM is an order of magnitude faster in wall-clock time (45.1s vs. 444.0s), proving its superior scalability.

Bug Cases. We responsibly disclosed these bug-triggering cases, with details of the three previously unknown bugs (Bug #3, #5, and #8), and all have been confirmed by developers.

Bug #3 and #5 share a similar trigger condition and root cause, both manifesting as a crash when the CC processes an input context pool in which certain contexts are not referenced by any constraint. Specifically, during CC core initialization, internal mappings from context names to contexts and from context names to constraints are established, but only those contexts referenced by at least one constraint are registered in the corresponding lookup tables. When the CC subsequently processes context changes (e.g., additions or modifications), a lookup for an unreferenced context returns `null`, and the absence of a null guard results in a null-pointer crash. The official patch for Bug #5 (below) confirms this diagnosis, whereas SOTA-Gao fails to trigger these crashes entirely.

Patch for Bug #5 found by CCM in MG_L.

```
   protected Map<...> checkChange(AddChange ac) {
     Map<...> res = new HashMap<>();
     String ts = ac.getTargetSet();
+    if (!sets.containsKey(ts) || !s2c.containsKey(ts)) {
+      s2c.put(ts, new HashSet<>());
+    }
     sets.get(ts).add(ac.getContext());
     Set<String> rel = s2c.get(ts);
     // ......
   }
```

Bug #8 is a concurrency flaw in INFUSE that requires an input constraint with at least three quantifiers and random element injection. SOTA-Gao cannot actively introduce such quantifier complexity, as it mostly preserves original structures. Therefore, it would have to randomly select one of only eight source constraints (out of over 1,000) that already have three quantifiers while also applying the specific MR that injects random elements, a coincidence virtually impossible within a 24-hour budget. In contrast, CCM's free PU composition and reconstruction enables far greater diversity and complexity in constraint structures, making the detection of such complex bugs straightforward and deterministic. The patch below resolves this concurrency issue.

Patch for Bug #8 found by CCM in $INFUSE_L$.

```
   switch (eF.getName()) {
     case "forall": // Bug also applies to "exists" case
       ...
-      patNodeMap.put(eF.attr("in"), new HashSet<>());
+      patNodeMap.put(eF.attr("in"),
+        ConcurrentHashMap.newKeySet());
       ...
   }
```

Answer to RQ3: *CCM demonstrates highly effective practical bug-detection capabilities, revealing all documented bugs and reporting three previously unknown ones in real-world CC implementations, while maintaining remarkable efficiency.*

D. Threats to Validity and Discussion

Internal Validity. The primary internal threat is the randomness of dynamic test generation. We mitigated this by repeating all experiments three times and applying rigorous non-parametric statistical tests. Additional threats include implementation correctness and hyperparameter selection. We used the official SOTA-Gao implementation, a standardized fail-fast pipeline, and hyperparameters aligned with prior literature [12]. The randomness injection mechanism adds immaterial context elements for diversity without altering CCM's core inference, which remains theoretically sound (Section III-F), and introduces negligible overhead that does not confound performance measurements.

Construct Validity. A major construct threat is whether artificial mutants reflect real-world faults. We addressed this by filtering out trivial and equivalent mutants and cross-validating against real-world bugs in RQ3. Threats to evaluation design were mitigated by adopting standard metrics (Mutation Score, Time2Bug, Case2Bug). For baselines, we selected SOTA-Gao (the only existing MT-based automated approach for CC) and Rand-Crash as a sanity check, ensuring a direct and state-of-the-art comparison.

External Validity. The primary external threat is the generalizability of our findings. While our evaluation covers six distinct versions of open-source CC systems (including the latest deployments of MG and INFUSE), generalization to domain-specific variants or proprietary CC middleware remains unexplored. To bootstrap test generation, we systematically enumerated valid input constraints for comprehensive FOL coverage; future work could mine constraints from industrial deployment logs to better mirror developer behaviors.

V. DISCUSSION

We discuss the application and generalizability of CCM.

Our experiments yield practical recommendations for deploying CCM effectively. First, we advise limiting PU composition to two rounds. In our configuration, two rounds on 1,452 initial PUs yield 3,429,492 candidates, providing comprehensive coverage of syntactic structures and transformation patterns (Fig. 4) within acceptable overhead. As the round number increases, the candidate pool grows exponentially. This limit reflects a deliberate trade-off between diversity and cost, and can be adjusted for different initial PU pool sizes.

Second, our ARS strategy disperses selected PUs across the structural space for efficient large-pool selection, with smaller batches improving dispersion at the cost of concurrency. Experiments across batch sizes from $1\times$ to $16\times$ at 2,000 cases show negligible mutation-score variation, while throughput improves 28.55% from $1\times$ to $4\times$ and plateaus thereafter. Thus, $4\times$ threads balances diversity and concurrency.

Beyond these configuration recommendations, the core philosophy of CCM, redefining metamorphic testing as a deconstruction-reconstruction pipeline, is transferable to other domains, such as compiler differential testing and search engine evaluation. However, practical adaptation requires domain-specific redesign of PUs and composition rules. We plan to explore these extensions in the future toward a generalized framework that lowers the barrier for practitioners.

VI. Related Work

Constraint checking is widely employed to maintain the consistency of software artifacts. This practice extends from XML documents [27]–[30] and UML models [31]–[33], to distributed source code [34]. With the proliferation of pervasive computing and human-cyber-physical systems (HCPS) [2], constraint checking has become essential for validating dynamic, real-time contexts. Such systems, including Industrial IoT (IIoT) platforms [35], [36] and autonomous driving systems [37], rely on checks to prevent abnormal behaviors caused by erroneous sensing or environmental noise. Consequently, various approaches [6]–[11] have been proposed to detect problematic contexts based on FOL consistency constraints. Beyond standalone approaches, middleware frameworks such as SEPAL [3] integrate constraint-checking services to ensure reliable behavioral adaptation.

Metamorphic testing (MT) is a prominent approach to mitigating the test oracle problem [20]. Its core principle is to leverage existing tests to generate new test cases and verify their outputs using metamorphic relations (MRs). MT has demonstrated significant efficacy in validating hard-to-test software systems across various domains, including search engines [38], SMT solvers [13], C compilers [14], [15], natural language processing pipelines [39], [40], large language models [16], [17], and content moderation software [41], [42].

Despite its effectiveness, traditional MT relies on manually crafted MRs, and their diversity is limited by the designer's domain knowledge. To overcome this, extensive research has explored *automated MR derivation*. Techniques range from using machine learning and search-based algorithms to infer likely MRs [43], [44], to performing MR composition by sequentially combining manually-defined base relations to formulate complex ones [45]. Recently, Large Language Models (LLMs) have also been leveraged to automate MR generation via prompt engineering [46].

However, these generalized techniques struggle with Constraint Checking (CC) systems. The state-of-the-art approach by Gao et al. [12] still depends on manually crafted MRs, inherently limiting test case diversity. To address this, our proposed Compositional Constructive Metamorphism (CCM) introduces an automated, constructive paradigm.

VII. Conclusion

This work studies the testing problem for constraint checking (CC) implementations, which are widely used but remain under-tested due to the absence of a test oracle. We propose Compositional Constructive Metamorphism (CCM), a novel perspective that reformulates metamorphic testing via iterative primitive unit decomposition and test case reconstruction with enforced diversity. CCM substantially outperforms existing baselines, improving mutation score by 17.9% (absolute) while reducing test case count by 84.7% and time cost by 26.6%. It also detects three previously unknown bugs in three intensively-tested real-world CC implementations. The CCM artifact, along with its supplementary resources, is publicly available at [21].

Acknowledgment

This research was supported by the Natural Science Foundation of China (Grant Nos. 92582201 and 62302209), the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (No. JYB2025XDXM118). The authors gratefully acknowledge the support from the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

References

[1] F. D. Macías-Escrivá, R. E. Haber, R. M. del Toro, and V. Hernández, "Self-adaptive systems: A survey of current approaches, research challenges and applications," *Expert Syst. Appl.*, vol. 40, no. 18, pp. 7267–7279, 2013.

[2] J. Zhou, Y. Zhou, B. Wang, and J. Zang, "Human-cyber-physical systems (HCPSs) in the context of new-generation intelligent manufacturing," *Engineering*, vol. 5, no. 4, pp. 624–636, 2019.

[3] S. Zhang, R. Yang, L. Zhang, H. Li, M. Wang, M. Gao, H. Hu, H. Wang, Y. Qin, and C. Xu, "SEPAL: A consistency-driven programming framework and runtime support for human-cyber-physical systems with reliable sensing and dynamic adaptation," *J. Comput. Sci. Technol.*, vol. 40, no. 4, pp. 958–968, 2025.

[4] A. Karkouch, H. Mousannif, H. Al Moatassime, and T. Noel, "Data quality in internet of things: A state-of-the-art survey," *J. Netw. Comput. Appl.*, vol. 73, pp. 57–81, 2016.

[5] K. Ni, N. Ramanathan, M. N. H. Chehade, L. Balzano, S. Nair, S. Zahedi, E. Kohler, G. J. Pottie, M. H. Hansen, and M. B. Srivastava, "Sensor network data fault types," *ACM Trans. Sens. Networks*, vol. 5, no. 3, pp. 25:1–25:29, 2009.

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

[7] C. Xu, Y. Liu, S. Cheung, C. Cao, and J. Lv, "Towards context consistency by concurrent checking for internetware applications," *Sci. China Inf. Sci.*, vol. 56, no. 8, pp. 1–20, 2013.

[8] J. Sui, C. Xu, W. Xi, Y. Jiang, C. Cao, X. Ma, and J. Lu, "GAIN: gpu-based constraint checking for context consistency," in *Proceedings of the 21st Asia-Pacific Software Engineering Conference, APSEC 2014, Jeju, South Korea, December 1-4, 2014. Volume 1: Research Papers*, S. S. Cha, Y. Guéhéneuc, and G. Kwon, Eds. IEEE Computer Society, 2014, pp. 319–326.

[9] B. Guo, H. Wang, C. Xu, and J. Lu, "GEAS: generic adaptive scheduling for high-efficiency context inconsistency detection," in *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*. IEEE Computer Society, 2017, pp. 137–147.

[10] C. Chen, H. Wang, L. Zhang, C. Xu, and P. Yu, "Minimizing link generation in constraint checking for context inconsistency detection," in *Proceedings of the IEEE 33rd International Symposium on Software Reliability Engineering, ISSRE 2022, Charlotte, NC, USA, October 31 - Nov. 3, 2022*. IEEE, 2022, pp. 13–24.

[11] L. Zhang, H. Wang, C. Xu, and P. Yu, "INFuse: Towards efficient context consistency by incremental-concurrent check fusion," in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution, ICSME 2022, Limassol, Cyprus, October 3-7, 2022*. IEEE, 2022, pp. 187–198.

[12] M. Gao, H. Wang, and C. Xu, "Testing constraint checking implementations via principled metamorphic transformations," in *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2024, Rovaniemi, Finland, March 12-15, 2024*. IEEE, 2024, pp. 884–895.

[13] D. Winterer, C. Zhang, and Z. Su, "Validating SMT solvers via semantic fusion," in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, A. F. Donaldson and E. Torlak, Eds. ACM, 2020, pp. 718–730.

[14] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014, Edinburgh, United Kingdom - June 09 - 11, 2014*, M. F. P. O'Boyle and K. Pingali, Eds. ACM, 2014, pp. 216–226.

[15] C. Sun, V. Le, and Z. Su, "Finding compiler bugs via live code mutation," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, E. Visser and Y. Smaragdakis, Eds. ACM, 2016, pp. 849–863.

[16] S. Cho, S. Ruberto, and V. Terragni, "LLMorph: Automated metamorphic testing of large language models," in *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering, ASE 2025, Seoul, Korea, Republic of, November 16-20, 2025*. IEEE, 2025, pp. 4102–4105.

[17] W. Wu, Y. Cao, N. Yi, R. Ou, and Z. Zheng, "Detecting and reducing the factual hallucinations of large language models with metamorphic testing," *Proc. ACM Softw. Eng.*, vol. 2, no. FSE, pp. 1432–1453, 2025.

[18] E. J. Weyuker, "On testing non-testable programs," *Comput. J.*, vol. 25, no. 4, pp. 465–470, 1982.

[19] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Trans. Software Eng.*, vol. 41, no. 5, pp. 507–525, 2015.

[20] T. Y. Chen, S. Cheung, and S. Yiu, "Metamorphic testing: A new approach for generating next test cases," *CoRR*, vol. abs/2002.12543, 2020.

[21] "Supplementary Materials," https://doi.org/10.5281/zenodo.21666913.

[22] "INFUSE," https://github.com/yuzi-zly/INFUSE.

[23] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "PIT: a practical mutation testing tool for java (demo)," in *Proceedings of the 25th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, A. Zeller and A. Roychoudhury, Eds. ACM, 2016, pp. 449–452.

[24] "CMID," https://github.com/growinware-cn/CMID.

[25] "MG," https://github.com/cychen2021/mg.

[26] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[27] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein, "xlinkit: a consistency checking and smart link generation service," *ACM Trans. Internet Techn.*, vol. 2, no. 2, pp. 151–185, 2002.

[28] C. Nentwich, W. Emmerich, A. Finkelstein, and E. Ellmer, "Flexible consistency checking," *ACM Trans. Softw. Eng. Methodol.*, vol. 12, no. 1, pp. 28–63, 2003.

[29] S. P. Reiss, "Incremental maintenance of software artifacts," *IEEE Trans. Software Eng.*, vol. 32, no. 9, pp. 682–697, 2006.

[30] H. A. H. Handley, W. Khallouli, J. Huang, W. Edmonson, and N. Kibret, "Maintaining the consistency of sysml model exports to XML metadata interchange (XMI)," in *Proceedings of the IEEE International Systems Conference, SysCon 2021, Vancouver, BC, Canada, April 15 - May 15, 2021*. IEEE, 2021, pp. 1–8.

[31] A. Egyed, "Instant consistency checking for the UML," in *Proceedings of the 28th IEEE/ACM International Conference on Software Engineering, ICSE 2006, Shanghai, China, May 20-28, 2006*, L. J. Osterweil, H. D. Rombach, and M. L. Soffa, Eds. ACM, 2006, pp. 381–390.

[32] R. S. Bashir, S. P. Lee, S. U. R. Khan, V. Chang, and S. Farid, "UML models consistency management: Guidelines for software quality manager," *Int. J. Inf. Manag.*, vol. 36, no. 6, pp. 883–899, 2016.

[33] H. Wen, J. Wu, J. Jiang, G. Tang, and Z. Hong, "A formal approach for consistency management in UML models," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 33, no. 5, pp. 733–763, 2023.

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

[35] R. Wang, X. Mou, T. Wo, M. Zhang, Y. Liu, T. Wang, P. Liu, J. Yan, and X. Liu, "ACbot: An IIoT platform for industrial robots," *Frontiers Comput. Sci.*, vol. 19, p. 194203, 2025.

[36] B. Yuan, K. Zheng, Y. Jia, J. Ren, K. Wang, S. Shi, D. Zou, and H. Jin, "DMCGuard: risky perils and fine-grained control on IoT multiple device management channels," *Frontiers Comput. Sci.*, vol. 20, p. 2005104, 2026.

[37] J. Chen, Y. Qin, H. Wang, and C. Xu, "Simulation might change your results: A comparison of context-aware system input validation in simulated and physical environments," *J. Comput. Sci. Technol.*, vol. 37, no. 1, pp. 83–105, 2022.

[38] Z. Q. Zhou, S. Xiang, and T. Y. Chen, "Metamorphic testing for software quality assessment: A study of search engines," *IEEE Trans. Software Eng.*, vol. 42, no. 3, pp. 264–284, 2016.

[39] Z. Liu, Y. Feng, Y. Yin, J. Sun, Z. Chen, and B. Xu, "QATest: A uniform fuzzing framework for question answering systems," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 2022, pp. 81:1–81:12.

[40] Z. Liu, Y. Feng, and Z. Chen, "DialTest: automated testing for recurrent-neural-network-driven dialogue systems," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2021, Virtual Event, Denmark, July 11-17, 2021*, C. Cadar and X. Zhang, Eds. ACM, 2021, pp. 115–126.

[41] W. Wang, J. Huang, W. Wu, J. Zhang, Y. Huang, S. Li, P. He, and M. R. Lyu, "MTTM: metamorphic testing for textual content moderation software," in *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 2387–2399.

[42] W. Wang, Y. Wu, J. Zhang, S. Li, Y. Peng, W. Chen, S. Wang, and M. R. Lyu, "Metamorphic testing for audio content moderation software," in *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering, ASE 2025, Seoul, Korea, Republic of, November 16-20, 2025*. IEEE, 2025, pp. 2906–2918.

[43] U. Kanewala and J. M. Bieman, "Using machine learning techniques to detect metamorphic relations for programs without test oracles," in *Proceedings of the IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013, Pasadena, CA, USA, November 4-7, 2013*. IEEE Computer Society, 2013, pp. 1–10.

[44] J. Zhang, J. Chen, D. Hao, Y. Xiong, B. Xie, L. Zhang, and H. Mei, "Search-based inference of polynomial metamorphic relations," in *Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering, ASE 2014, Vasteras, Sweden - September 15 - 19, 2014*, I. Crnkovic, M. Chechik, and P. Grünbacher, Eds. ACM, 2014, pp. 701–712.

[45] H. Liu, X. Liu, and T. Y. Chen, "A new method for constructing metamorphic relations," in *Proceedings of the 2012 12th International Conference on Quality Software, Xi'an, Shaanxi, China, August 27-29, 2012*, A. Tang and H. Muccini, Eds. IEEE, 2012, pp. 59–68.

[46] Y. Zhang, D. Towey, and M. Pike, "Automated metamorphic-relation generation with ChatGPT: An experience report," in *Proceedings of the 47th IEEE Annual Computers, Software, and Applications Conference, COMPSAC 2023, Torino, Italy, June 26-30, 2023*, H. Shahriar, Y. Teranishi, A. Cuzzocrea, M. Sharmin, D. Towey, A. K. M. J. A. Majumder, H. Kashiwazaki, J. Yang, M. Takemoto, N. Sakib, R. Banno, and S. I. Ahamed, Eds. IEEE, 2023, pp. 1780–1785.