

Testing Constraint Checking Implementations via Principled Metamorphic Transformations

Mingchen Gao^{†‡}, Huiyan Wang^{*†‡}, Chang Xu^{†‡}

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

[‡]Department of Computer Science and Technology, Nanjing University, Nanjing, China

gmc@smail.nju.edu.cn, {why, changxu}@nju.edu.cn

Abstract—Constraint checking techniques are being widely used for ensuring the consistency of software artifacts during their development and evolution (e.g., detecting inconsistency in an application’s running contexts or identifying rule violation in the code being developed). Typically, consistency constraints are formulated and checked upon the changes of software artifacts under checking. When any constraint is violated, an inconsistency is said to occur and then follow-up actions can be taken to remedy the problem. Currently, various constraint checking techniques have been proposed and implemented with sophisticated mechanisms for higher efficiency and scalability. However, these implementations could be far from being well tested due to their oracle problems, i.e., hardly able to tell what the checking result should be, given any software artifacts and their consistency constraints to check. In this paper, we leverage metamorphic testing and propose a family of metamorphic relations catered for testing constraint checking implementations. We dedicatedly design these relations by following the sensitivity principle via a fine-granularity control and the diversity principle via input-oriented transformations. Our experiments reported promising results (disclosing 80% mutation bugs and five real bugs) without the need of any manual labeling.

Index Terms—Constraint checking, metamorphic relation, principled transformation

I. INTRODUCTION

The capability of constraint checking is commonly required by many quality applications. They rely on the capability to deal with their ever-changing environmental data and make proper adaptations, e.g., applications on sensor networks [1]–[3], IoT (Internet of Things) [4], [5] and ITS (Intelligent Transportation Systems) [6]. Consider a scenario in which an application makes adaptations according to its sensed environmental changes. It derives high-level interpretations of its environments (named *contexts*) to support its adaptations, but the derived contexts have to be consistent. Otherwise, it could probably behave abnormally or even crash [7], [8]. Currently, various constraint checking techniques have been proposed and implemented (named *CC implementations*). They exploit sophisticated mechanisms to cope with varying efficiency or scalability concerns, and make timely responses when their checked consistency constraints are violated, so that the concerned applications could be timely guarded from abnormal executions. Taking the adaptive computing field for example, various CC implementations have been realized to guard the consistency of applications’ contexts, e.g., ECC and PCC

(entire and partial checking for context consistency) [8], GEAS (grouped checking for context consistency) [7], INFUSE (parallel and incremental context checking) [9], and MG (reducing link generation redundancy in context checking) [10].

However, the CC implementations could be far from being well tested, due to the complex logics in constraint checking and the lack of a precise oracle [11] that decides each checking result (considering any software artifacts under checking (i.e., *data part*) and any consistency constraints to enforce on the artifacts (i.e., *constraint part*)). Consider a smart route guidance application [7], [12], with frequent changes in its collected contexts (e.g., GPS coordinates, driving velocities, etc.). One can hardly tell what a CC implementation should report at any moment, given an arbitrary consistency constraint on these data (e.g., special relationships among GPS coordinates and driving velocities concerning a vehicle’s nearby ones). Existing research either referred to a well-defined dataset that already incorporated manual result labels (e.g., *truth values* that indicate constraint satisfaction or violation, and *links* [8] that explain what causes the satisfaction or violation), or relied on differential testing by cross validation among different CC implementations. However, these workarounds inevitably limit the test automation (needing costly manual labeling) and completeness (different CC implementations could have different functionalities). Therefore, this strongly calls for a new testing approach that can *automatically* generate test inputs and validate checking results *without* manual labeling, which is highly desirable for this line of research.

One promising approach to addressing the oracle problem is *metamorphic testing* [13]. It alleviates the need for specifying explicit oracles for non-testable or hard-to-test programs, e.g., search implementation [14], SMT solvers [15], C compilers [16], [17], natural language translators [18], [19], autonomous cars [20], [21], etc. Metamorphic testing realizes this by modeling and validating properties among multiple program executions, which are known as *metamorphic relations* (MRs). However, figuring out metamorphic relations specific to target software systems requires high intelligence, and existing MRs can hardly apply to the testing of CC implementations. Particularly, we question, *how the inputs to CC implementations (i.e., the data under checking and the constraints to enforce) should be transformed, such that the outputs from CC implementations can be predicted*. This remains unclear and the answer seems non-trivial.

*Corresponding author

In this paper, we propose and realize two principles in the design of MRs specific for testing CC implementations, namely, *sensitivity principle* (existing work [16], [22] advocated sensitive MRs that exercise specific functionalities triggered by certain program executions), and *diversity principle* (existing work [23]–[25] suggested diverse MRs that exercise different parts in a program to expose potential bugs).

With the two design principles, we tackled the difficulty of designing MRs and propose a family of MRs for testing CC implementations. For the sensitivity, we conduct a fine-granularity analysis that measures the impacts of values of the data under checking and the constraints to enforce, with respect to each formula type in a consistency constraint, on the checking outputs. For the diversity, we consider different input-oriented transformations that enumerate all parts in CC implementations’ inputs and their combinations, i.e., transforming the data under checking, the constraints to enforce, and both at the same time. Altogether, we designed 15 MRs, to make their utilization to be sensitive and diverse in testing CC implementations. Our experimental results report that: (1) these new MRs detected around 80% mutants, with each MR exhibiting unique effectiveness in killing certain mutant types; (2) these MRs also detected five realistic bugs in three CC implementations, which had remained undetected for long time (months to years). We note that this was achieved without the need of any manual result labeling in constraint checking, and thus is very desirable. To facilitate further research, we also release our artifact and detailed documentation online [26].

The remainder of this paper is organized as follows. Section II introduces background. Section III presents our principled CC implementation testing methodology. Section IV evaluates our methodology for MRs’ effectiveness in testing real-world CC implementations. Section V discusses related work in recent years, and finally Section VI concludes this paper.

II. BACKGROUND

A. A Typical Application and Research Review

Open-environment adaptive applications have gained popularity in many fields due to smart adaptations to open environments, which highly depend on the quality of collected environmental contexts at runtime. To maintain high-quality contexts, constraint checking implementations are usually integrated into a context-management middleware between environment and application layers by structure, as shown in Fig. 1, to check and accordingly resolve to ensure high-quality contexts before feeding them into the application.

Here, we use a highway toll system, a typical ITS (Intelligent Transportation System) [6], to illustrate how a constraint checking technique can be integrated as a context management middleware to maintain the quality of applications’ contexts. The depends on car-tracking contexts collected by gantries along the highway to calculate tolls and accordingly charge drivers. However, the quality of collected contexts can be easily affected by many factors, e.g., traffic noises, gantry interferences, etc, thus leading to severely problematic contexts [27] and can cause wrong or even outrageous toll

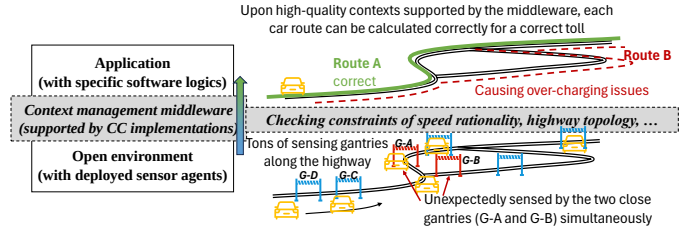


Fig. 1. Exhibition for the context management middleware supported by constraint checking techniques

calculations. For example, as shown in Fig. 1, the yellow car has been unexpectedly tracked by gantry A and B simultaneously due to their too-close locations, usually known as common cross-read issues in sensing fields [28]–[30]. If such contexts are directly fed into the application without any context management, it would cause an over-charging issue when accidentally considering route B.

To identify and resolve such problematic contexts, constraint checking techniques are typically integrated into the middleware between the environment and application layers. It systematically checks certain properties for the application, e.g., gantry continuity, speed rationality, highway topology knowledge, etc. For example, when supported with context management considering highway topology information by constraint checking, cars passing gantries A and B simultaneously should be identified as problematic in time, and then resolved before feeding into the application to avoid application misbehaviors. It is also worth mentioning that constraint checking can be applied not only in such a highway toll system but also in any system that processes dynamically-changing data with specific constraints or properties.

To support such middleware infrastructures, in the last ten years, many researchers have developed a variety of different constraint checking tool prototypes, provided with different degrees of checking quality and velocity, e.g., CMID [31], MG [10], INFUSE [32], etc. However, due to the complex logic in constraint checking and the lack of a precise oracle that decides the checking result, these CC implementations are far from being well tested. Commonly, developers only claim to test their implementations upon manually validated but limited datasets, and to our knowledge, there is no previous work on testing such CC implementations effectively. In this paper, we propose to leverage ideas from metamorphic testing which has exhibited nice effectiveness in testing non-oracle software artifacts [15], [16], and test CC implementations guided by a series of effective metamorphic relations with principled transformations.

B. Preliminary

Typically, a constraint checking implementation takes two inputs: (1) *data under checking*, and (2) a *constraint for checking*. We define some concepts first.

Data under checking is a collection of structured information about the data interesting to checking [7], which can be modeled as a finite set of elements, each denoting a relevant

part of this pattern. *Constraints for checking* are formulated for property checking, and can be typically specified using the following first-order logical (FOL) formulae [7], [8], [33]–[38]. Here, P is a pattern which matches a piece of aspects from the data under checking, v_i is a variable that takes an element from P as its value; terminal *bfunc* is an checking-specific predicate that returns a Boolean value (T or F):

$$\begin{aligned} f ::= & \forall v \in P(f) \mid \exists v \in P(f) \mid (f) \text{ and } (f) \mid \\ & (f) \text{ or } (f) \mid (f) \text{ implies } (f) \mid \text{not } (f) \mid \\ & \text{bfunc}(v_1, v_2, \dots, v_n) \mid \text{T} \mid \text{F}. \end{aligned}$$

For the highway toll system as shown in Fig. 1, suppose the system is now tracking cars on the highway according to its latest and previous gantries passed. Suppose only car c_1 , which has passed gantry D first and C next. Contexts that are interesting would be now modeled as patterns: $P_{\text{latest}} = \{r_{1C}\}$ and $P_{\text{previous}} = \{r_{1D}\}$, denoting car records indicating c_1 just passed gantry C from gantry D. We use subscripts to denote a record with respect to a specific car and the sensing gantry, e.g. record r_{1C} denoting car c_1 passing gantry C. When the time flows and car c_1 is sensed by gantry A. Patterns would be updated accordingly: $P_{\text{latest}} = \{r_{1A}\}$ and $P_{\text{previous}} = \{r_{1C}\}$. However, due to sensing noises, gantry B also sensed the same car in a quite short period, thus leading to the current patterns being unexpectedly updated: $P_{\text{latest}} = \{r_{1B}\}$ and $P_{\text{previous}} = \{r_{1A}\}$. Such patterns indicate that car c_1 was last sensed by gantry A and is now sensed by gantry B, which is actually impossible because gantry A and gantry B are on different path branches in the highway topology.

To detect such cross-read issues for highway contexts, one may formulate the continuity property as follows:

$$S_{\text{loc}} ::= \forall v_1 \in P_{\text{latest}} (\text{not}(\exists v_2 \in P_{\text{previous}} (\text{cross}(v_1, v_2)))).$$

where $\text{cross}(v_1, v_2)$ is used to check whether two recorded gantries relate to cross branches in the highway topology, a.k.a., uncontinuous. It returns the value T when the records denoted by v_1 and v_2 represent the same car and the associated gantries are truly on cross branches, otherwise, F.

In this example, data under checking now denotes the obtained $D = \{P_{\text{latest}}, P_{\text{previous}}\}$ and the constraint for checking is the formulated consistency constraint S_{loc} . Then, by constraint checking, one can observe that constraint S_{loc} is clearly violated when $P_{\text{latest}} = \{r_{1B}\}$ and $P_{\text{previous}} = \{r_{1A}\}$, since gantries A and B are in different cross branches. We next elaborate on how constraint checking is usually conducted.

C. Constraint Checking

Normally, any constraint for checking can be expanded to a *Consistency Computation Tree (CCT)* based on both its static constraint structure and the current data under checking. For example, consider the aforementioned consistency constraint S_{loc} and data D_1 containing $P_{\text{latest}} = \{r_{1B}, r_{2C}\}$ and $P_{\text{previous}} = \{r_{1A}, r_{2D}\}$, indicating that car c_1 was last sensed by gantry A and is now sensed by gantry B and car c_2 was last sensed by gantry D and is now sensed by gantry C. Then CCT would be expanded as shown in Fig. 2.

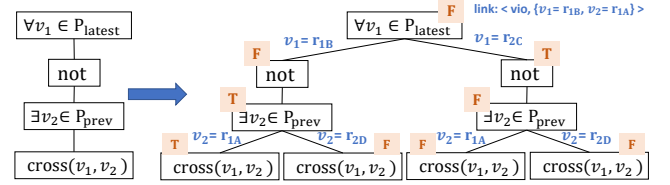


Fig. 2. CCT expansion for constraint S_{loc}

Note that any node representing either a universal or existential quantifier would be expanded with sub-structures corresponding to different elements in its associated pattern in the constraint. We also annotate the CCT with variable assignments in expansion for better illustration.

Typically, a constraint checking process would be conducted based on such a CCT-alike structure as in Fig. 2 and includes two phases, i.e., truth value evaluation and link generation. Truth value evaluation would evaluate the whole CCT recursively to obtain the root's truth value (T or F) based on evaluating each associated *bfunc* leaf node. The whole truth value evaluation is usually conducted in a bottom-up manner following a post-order traversal. In this case, cross function would return T only when $v_1 = r_{1B}$ and $v_2 = r_{1A}$, since now the same car c_1 passed two uncontinuous gantries B and A. When all necessary nodes are evaluated, their parent nodes can then be evaluated. We also annotate the truth values for all nodes in Fig. 2, and one can observe that the whole constraint is now evaluated to be F (as the CCT root's truth value is).

Besides the truth value evaluation, to facilitate the following context resolution or event-driven decision, constraint checking usually deploys link generation to explain why any consistency constraint is satisfied or violated and which specific elements are concerned for the constraint satisfaction or violation. The link generation is similarly conducted in a bottom-up manner and one can refer to existing work [8] [7] for detailed link generation semantics. For the example in Fig. 2, the final link result contains one link, i.e., $\langle \text{vio}, \{v_1 = r_{1B}, v_2 = r_{1A}\} \rangle$, indicating that this constraint is violated due to r_{1B} in P_{latest} and r_{1A} being in P_{previous} (as reported by a “vio” link).

A *constraint checking (CC) implementation* usually takes two inputs: (1) *data under checking (D)* containing all patterns with structured information interesting to the checking, and (2) a *constraint for checking (s)*, a well-formed FOL formula to be checked. Then, it would produce two outputs: (1) a *truth value (Rt)* denoting whether the whole constraint is satisfied or not concerning the current data, and possibly (2) a set of *link explanations (Rl)* containing links explaining how the constraint is satisfied or violated in specific. That is, $(D \times s) \xrightarrow{CC} (Rt \times Rl)$. For simplicity, we model a CC implementation as such without losing any generalizability, since CC implementations typically check constraints independently even when multiple constraints are fed. For example, when feeding D and S_{loc} as shown in Fig. 2, it would produce a truth value F and a link set $\langle \text{vio}, \{v_1 = r_{1B}, v_2 = r_{1A}\} \rangle$.

D. Metamorphic Testing

Metamorphic testing is an effective testing mechanism, especially for software artifacts without testing oracles. In metamorphic testing, it checks important properties captured by metamorphic relations (MR).

Formally, consider the software under test (SUT), which can produce output as $y_0 = f(x_0)$ for any input x_0 . A metamorphic relation includes two parts: a *transformation function* $\mathcal{T}$ explaining how to obtain a follow-up test input x_1 for the given source test (i.e., source input x_0), i.e., $x_1 = \mathcal{T}(x_0)$, and an *assertion function* $\mathcal{A}$ explaining how to examine whether the follow-up test output is expected or not, i.e., $\mathcal{A}(y_0, y_1) = T|F$. As shown in Fig. 3, metamorphic testing with $MR = \{\mathcal{T}, \mathcal{A}\}$ is typically conducted as follows.

- 1) Execute SUT with source input x_0 to obtain output y_0 (i.e., a *source execution*);
- 2) Obtain follow-up input x_1 using the transformation function $\mathcal{T}$ designed in MR;
- 3) Execute SUT with follow-up input x_1 to obtain y_1 (i.e., a *follow-up execution*);
- 4) Examine whether any violation occurs based on the assertion function $\mathcal{A}$ designed in MR.

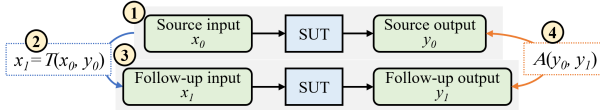


Fig. 3. Steps for metamorphic testing

If any violation is revealed, SUT is observed with the bug and some following debugging phases should be conducted accordingly. Although we here explain metamorphic relations using numerical properties that arithmetic equations can capture, MRs can be much more general and have exhibited great testing effectiveness in many fields [15]–[18], [21].

III. METHODOLOGY

A. Principled Metamorphic Testing for CC implementations

In order to test constraint checking implementations, we propose a principled metamorphic testing method. Basically, we raise two principles to guide our whole design of metamorphic relations, i.e., *sensitivity* and *diversity*.

To meet the sensitivity principle, we aim to design MRs that focus on specific program functionality in a fine-granularity way, especially requesting our designed transformation in each MR to be restricted by especially considering a certain output situation. Considering the natural output truth value for CC implementations (T or F), we aim to design our transformation and evaluate how each transformation would impact on the constraint checking output (e.g., truth value being T or F). By doing so, we can design MRs that are provided with a fine-granularity control on its transformation impact and avoid being too general that all program executions (no matter which truth value is produced and expected in the execution) should obey.

- *T-dependent*: Transformation only applied to source executions restricted by the output truth value being T.
- *F-dependent*: Transformation only applied to source executions restricted by the output truth value being F.

To meet the diversity principle, we aim to design MRs that are different from construction. Therefore, we distinguish which input is oriented when designing the key transformation functions in metamorphic relations, and thus aim to design transformation functions $\mathcal{T}$ for CC implementations that only allow transforming certain input parts. Note that, a constraint checking implementation has two typical inputs: data under checking and a constraint for checking, and produces its outputs: a truth value, and a set of link explanations. Therefore, considering both any isolated input and input combination, there are three orientations to consider in designing the key transformation functions as follows.

- *Data-oriented*: Only transform the input data;
- *Cons-oriented*: Only transform the input constraint;
- *All-oriented*: Transform both the data and constraint.

By distinguishing different input orientations in metamorphic testing, we expect to design naturally-diverse metamorphic relations from construction. For systems with more complex inputs, one can also leverage such an input-oriented designing routine when designing their own MRs.

B. Derived Principled Metamorphic Relations

Consider a CC implementation that takes the inputted data D and FOL constraint s , and produces its checking result including a truth value Rt , and a set Rl of link explanations. Suppose the source test input being D_0 and s_0 , and source output being Rt_0 and Rl_0 , i.e., $(Rt_0, Rl_0) = CC(D_0, s_0)$. We introduce our derived metamorphic relations with different input orientations for pursuing the diversity, i.e., data-oriented, constraint-oriented, and all-oriented, and different output dependencies for pursuing the sensitivity, i.e., T-dependent and F-dependent. We give the derived MRs in Table I.

1) *Data-oriented MR*: To design metamorphic relations transforming only the data under checking for CC implementations, we need to construct MRs that only transform the data under checking and meanwhile, the final constraint checking result can still be controlled. We leverage the inconsistency-incurring/hidden impact analyzed in existing work [7] (originally named as inc+/inc−), which explains how a change on the data under checking might affect the constraint checking result corresponding to a specific constraint. The inconsistency-incurring changes lead to more possibilities for incurring violation and inconsistency-hidden changes have the opposite effect. These two kinds of impacts of data changes depend only on the constraint itself and can be derived from the constraint statically. For example, constraint S_{loc} has two inconsistency-incurring changes $\langle +, P_{latest} \rangle$ and $\langle +, P_{previous} \rangle$, which means if we add any element to P_{latest} or $P_{previous}$, constraint would tend to generate more violations. By restricting different outputs considered in designing the transformation functions, we can then measure how changes

TABLE I
DERIVED PRINCIPLED METAMORPHIC RELATIONS

No	$\mathcal{T}$	$\mathcal{A}$	Description
MR-Data 1	if $Rt_0 == F$, $D_1 \leftarrow D_0$ with inc+ changes	$Rt_1 == Rt_0 \wedge ex(Rl_1, Rl_0)^*$	Transform the data under checking with more possibilities for incurring violations.
MR-Data 2	if $Rt_0 == T$, $D_1 \leftarrow D_0$ with inc- changes	$Rt_1 == Rt_0 \wedge ex(Rl_1, Rl_0)$	Transform the data under checking with more possibilities for incurring satisfactions.
MR-Cons 1	if $Rt_0 == T \wedge s_0$'s root is $\forall$, $\forall \leftarrow \exists$	$Rt_1 == Rt_0 \wedge ex(Rl_1, Rl_0)$	Transform the constraint from $\forall$ to $\exists$.
MR-Cons 2	if $Rt_0 == F \wedge s_0$'s root is $\exists$, $\exists \leftarrow \forall$	$Rt_1 == Rt_0 \wedge ex(Rl_1, Rl_0)$	Transform the constraint from $\exists$ to $\forall$.
MR-Cons 3	if $Rt_0 == T \wedge s_0$'s root is and, and $\leftarrow$ or	$Rt_1 == Rt_0 \wedge ex(Rl_1, Rl_0)$	Transform the constraint from and to or.
MR-Cons 4	if $Rt_0 == F \wedge s_0$'s root is or, or $\leftarrow$ and	$Rt_1 == Rt_0 \wedge ex(Rl_1, Rl_0)$	Transform the constraint from or to and.
MR-Cons 5	if $Rt_0 == F \wedge s_0$'s root is or, or $\leftarrow$ implies	$Rt_1 == T \wedge ex(Rl_1, Rl_0)$	Transform the constraint from or to implies.
MR-Cons 6	if $Rt_0 == F \wedge s_0$'s root is implies, implies $\leftarrow$ or	$Rt_1 == T \wedge ex(Rl_1, Rl_0)$	Transform the constraint from implies to or.
MR-Cons 7	if $Rt_0 == T \wedge s_0$'s root is and, and $\leftarrow$ implies	$Rt_1 == Rt_0 \wedge ex(Rl_1, Rl_0)$	Transform the constraint from and to implies.
MR-Cons 8	if $Rt_0 == F \wedge s_0$'s root is implies, implies $\leftarrow$ and	$Rt_1 == Rt_0 \wedge ex(Rl_1, Rl_0)$	Transform the constraint from implies to and.
MR-Cons 9	if $Rt_0 == F \wedge s_0$'s root is implies, switch implies's two sub-formulas	$Rt_1 == T \wedge ex(Rl_1, Rl_0)$	Transform the constraint from f_1 implies f_2 to f_2 implies f_1 .
MR-All 1	if $Rt_0 == F$, link s_0 's positive quantifier with and or negative one with or, $D_1 \leftarrow D_0$ with new patterns	$Rt_1 == Rt_0 \wedge ex(Rl_1, Rl_0)$	Transform both the constraint with a newly added and/or and data with new patterns.
MR-All 2	if $Rt_0 == T$, link s_0 's positive quantifier with or or negative one with and, $D_1 \leftarrow D_0$ with new patterns	$Rt_1 == Rt_0 \wedge ex(Rl_1, Rl_0)$	Transform both the constraint with a newly added and/or and data with new patterns.
MR-CCT 1	if $Rt_0 == F$, $D_1 \leftarrow D_0$ with inc+ changes	$Rt_1 == Rt_0 \wedge ex(SCCT_1, SCCT_0)$	Transform the data under checking with more possibilities for incurring inconsistencies.
MR-CCT 2	if $Rt_0 == F$, link s_0 's positive quantifier with and or negative one with or, $D_1 \leftarrow D_0$ with new patterns	$Rt_1 == Rt_0 \wedge ex(SCCT_1, SCCT_0)$	Transform both the constraint with a newly added and/or and data with new patterns.

*denotes each MR's assertion examination function $ex()$ for concerned outputs other than the truth value, i.e., link explanations (i.e., Rl_1 and Rl_0), or substantial CCTs (i.e., $SCCT_1$ and $SCCT_0$). For example, for MR-Data 1, its $ex()$ judges whether $Rl_1 \supseteq Rl_0$ is satisfied as mentioned in Section III.B. More detailed specifications for other MR's assertion examination functions can be found on [26].

would perform and impact the final output. Therefore, we design two data-oriented MRs that only transform the source data under checking D_0 to obtain follow-up data D_1 in testing and belong to T-dependent and F-dependent respectively.

MR-Data 1 (F-dependent): If in the source execution, s_0 is violated with Rt_0 being F, any change with an inconsistency-incurring impact on s_0 can be applied to data D_0 to obtain follow-up data D_1 . Then, in the follow-up execution, the corresponding Rt_1 would be the same as Rt_0 and Rl_1 would contain all links in Rl_0 (i.e., $Rl_1 \supseteq Rl_0$).

MR-Data 2 (T-dependent): If in the source execution, s_0 is satisfied with Rt_0 being T, any change with an inconsistency-hidden impact on s_0 can be applied to data D_0 to obtain follow-up data D_1 . Then, in the follow-up execution, the corresponding Rt_1 would be the same as Rt_0 and Rl_1 would contain all links in Rl_0 (i.e., $Rl_1 \supseteq Rl_0$).

We take MR-Data 1 as an example, and MR-Data 2 can be similarly analyzed. Note that, according to the definition [7], any inc+ change upon D_0 with respect to s_0 , can only incur new inconsistencies but can never lead to any existing inconsistencies undetectable. Therefore, when D_0 is transformed to D_1 with only the inc+ change applied, then in the follow-up execution, we can measure that the impact of such transformation can only lead to the whole constraint been violated with Rt_1 being F when Rt_0 being F, and the detected inconsistencies should contain all reported links in the source execution, i.e., $Rl_1 \supseteq Rl_0$. For example, for any constraint

with the form of $\forall v \in P(f)$, adding any element to the pattern P can only potentially incur new inconsistencies (thus possibly new violated links to report). Therefore, with respect to s_0 , one can easily construct follow-up inputs by only transforming data under checking D_0 to D_1 with more elements being added into pattern P if MR-Data 1 is considered for testing.

2) *Constraint-oriented MR*: Instead of transforming the data part for CC implementations, we can transform the constraint part only in designing transformation. In order to design MRs transforming only the input constraint, we need to construct MRs transforming the constraint only, that can make the final constraint checking result still able to be controlled. In order to pursue the sensitivity principle, we at the same time consider transforming different quantifiers (i.e., $\forall$, $\exists$, and to or, and implies) and measure how such quantifier transformation would impact on the whole constraint checking in a fine-granularity control. We also control different output dependencies (i.e., T/F-dependent) in the design.

MR-Cons 1 (T-dependent): If in the source execution, s_0 is satisfied (Rt_0 being T) and its *root quantifier* (the quantifier of its CCT root) is $\forall$, one can transform s_0 by changing its root quantifier from $\forall$ to $\exists$. Then, the follow-up truth value should be the same (Rt_1 being T) and links should include all satisfied links each exactly relating to a specific element in P.

In this case, s_0 is satisfied originally with its root quantifier being $\forall$, this means that all its children sub-trees of the CCT root should be evaluated to be T. Then, when the constraint

s_0 is transformed with its root quantifier changed from $\forall$ to $\exists$, its impact would be letting all the sub-trees with each's true evaluation being T stay the same and thus now make the whole constraint to be still T according to the semantics of $\exists$. More concretely, in this case, the follow-up links should contain satisfaction links each relating to a specific element in the corresponding pattern, which together explain how the constraint is now satisfied.

We can also similarly design to transform s_0 's root quantifier from $\exists$ to $\forall$ when s_0 is violated originally (Rt_0 being F). Then, the follow-up truth value result should also be the same (Rt_1 being F), and thus obtained the following MR-Cons 2.

MR-Cons 2 (F-dependent): If in the source execution, s_0 is violated (Rt_0 being F) and its root quantifier is $\exists$, one can transform its root quantifier to $\forall$ and expect the truth value result to be the same (Rt_1 being F) and links should include all violated links each relating to a specific element in P.

When s_0 's root quantifier is neither $\exists$ nor $\forall$, we transform quantifiers by switching among and, or, and implies as follows.

MR-Cons 3 (T-dependent): If in the source execution, s_0 is satisfied (Rt_0 being T) and its root quantifier is and, one can transform its root quantifier to or and expect the truth value result to be the same (Rt_1 being T).

In this case, due to the close semantics of or and and, MR-Cons 3 can be easy to follow. We can similarly transform s_0 's root quantifier from or to and, and to implies, implies to and, implies to or, and or to implies, and thus similarly obtain **MR-Cons 4-8** as shown in Table I.

Instead of transforming the quantifier, we consider transforming s_0 's inner structure. For example, in FOL language, implies's left and right formulas are naturally with different roles. We consider switching them as follows and the corresponding impact can also be controlled.

MR-Cons 9 (F-dependent): If in the source execution, s_0 is violated (Rt_0 being F) is with root quantifier of implies, one can exchange its two children formula under implies and expect the truth value result to be different (Rt_1 being T).

To save space, we neglect some details here. For example, to make such MRs stronger, we also further request assertions between source and follow-up link results for all MRs. Interested readers can reach our MR documentation [26].

3) *All-oriented MR*: We also design metamorphic relations that transform both the data part and constraint part at the same time for the CC implementation. To control the transformation impact, we control to insert a new formula into s_0 where the new formula has its own new patterns that need to be added into the data part. This leads to a simultaneous transformation of the data part and constraint part. In this way, we can transform both s_0 and D_0 for s_1 and D_1 and make its impact to be still controlled. Similarly, we design transformations of T-dependent and F-dependent respectively.

Note that it can be somehow difficult to measure the impact when connecting the original formula with any new formula, because one can not determine whether and how such a connection leads to violation/satisfaction of the whole constraint. To control the impact of such a connection with

respect to the violation or satisfaction of the whole constraint, we divide a constraint's associated quantifiers into two parts: *positive* and *negative*. A positive quantifier supports the root's violation or satisfaction while a negative quantifier supports it in the opposite way. In definitions, only the quantifier that is transitively through implies from the left or not from the root can reverse its supporting direction initialized to be positive. We thus design MR-All 1 and MR-All 2 as follows.

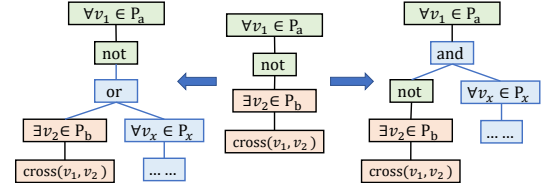


Fig. 4. Constraint transformation for MR-All 1

MR-All 1 (F-dependent): If in the source execution, s_0 is violated (Rt_0 being F), one can transform s_0 by linking one of its positive quantifiers with and or linking one of its negative quantifiers with or. Then, the follow-up truth value result should be the same (Rt_1 being F), and Rl_1 would contain all links in Rl_0 (i.e., $Rl_1 \supseteq Rl_0$).

As shown in Fig. 4, considering s_0 is originally violated, we can accordingly control to only link one of s_0 's positive quantifiers (colored by green) with and or any of its negative quantifiers (colored by orange) with or. In this way, the corresponding impact would be preserving all the original parts from s_0 that can thus support and dominate the whole s_1 's result to be the same as the original. In this way, s_1 should also be violated. Moreover, since a new formula is now linked by either and or or quantifier, more data associated with the newly linked pattern part should also be added to the original data under checking D_0 , and thus D_0 is also transformed to a new one, i.e., D_1 . Therefore, for MR-All 1, both s_0 and D_0 are transformed to obtain the follow-up inputs s_1 and D_1 , and the whole impact can also be controlled. We also similarly design MR-All 2 when s_0 is satisfied on the other hand.

MR-All 2 (T-dependent): If in the source execution, s_0 is satisfied (Rt_0 being T), one can transform s_0 by linking any of its positive quantifiers with or or linking any of its negative quantifiers with and. Then, the follow-up truth value result should be the same (Rt_1 being T) and Rl_1 would contain all links in Rl_0 (i.e., $Rl_1 \supseteq Rl_0$).

Guided by our sensitivity and diversity principles, all our designed MRs are given in Table I. Note that all these MRs are designed by assuming only inputs and outputs for CC implementations are available. If more internal states can be observed, e.g., CCT status like substantial parts accessible in existing work [10], we can additionally design MRs with respect to measuring the impact for such detailed CCT status under different transformations between source and follow-up executions. For example, in some work, substantial parts in a CCT, a.k.a., substantial CCT or SCCT, are accessible as an additional output when the constraint is eventually violated (naturally F-dependent). This structure links all tree nodes for

traced link explanations from the root when the constraint is violated, e.g., the sub-tree traced from the root, to $v_1 = r_{1B}$, and to $v_2 = r_{1A}$ in Fig. 2. Then, we can additionally design the following two MRs with similar principled transformations.

MR-CCT 1: If in the source execution, s_0 is violated with Rt_0 being F, any change with an inconsistency-incurring effect on s_0 can be applied to data D_0 to obtain follow-up data D_1 . Then, in the follow-up execution, the corresponding Rt_1 would be the same as Rt_0 and $SCCT_1$ would contain all nodes in $SCCT_0$ ($SCCT_1 \supseteq SCCT_0$).

MR-CCT 2: If in the source execution, s_0 is violated (Rt_0 being F), one can transform s_0 by linking any of its positive quantifiers with and or linking any of its negative quantifiers with or. in the follow-up execution, the corresponding Rt_1 would be the same as Rt_0 and $SCCT_1$ would contain all nodes in $SCCT_0$ ($SCCT_1 \supseteq SCCT_0$).

In summary, our designed MRs meet both sensitivity principles by considering different output situations and diversity principles by considering all three enumerated input orientations for the two inputs of CC implementations. We expect that these MRs can be effective in revealing violations for bug detection for CC implementations. Due to the page limit, we do not give all details. Interested readers can reach out to our documentation [26].

C. Source Input Synthesis and Follow-up Generation

To test CC implementations with the metamorphic relations, we prepare source inputs and generate follow-up ones below.

First, for synthesizing a consistency constraint, we leverage the idea from existing work [10] to synthesize well-formed FOL constraints under a certain height. For each constraint, we simulate the data under checking to contain multiple elements in each of its associated patterns with random values. Note that, as all constraints are synthesized, *bfunc* terminals do not carry real semantics. Therefore, to facilitate the testing, we can simulate their value calculations by enforcing the elements in any concerned *bfunc* carrying consistent values and exhaustively choosing all possible values when repeating different runs of constraint checking. By doing so, we obtain a synthesized source input, including a constraint s_0 and corresponding data under checking D_0 with all elements giving consistent *bfunc* values requested for checking. Be aware that, for each constraint, we exhaustively enumerate every possible value of each *bfunc*, and so actually obtain an exponential number of synthesized source inputs for testing. Then, we feed it to any constraint checking implementation to obtain its corresponding source output including both a truth value and a link set (as the source execution shows in Fig. 3).

Then, we can apply any of our designed MRs to generate a follow-up test input based on the source input according to each MR's specification. That is, if any MR in the MR-Data category is chosen, in follow-up execution, only D_0 would be transformed to generate D_1 and s_1 would be equal to s_0 . If any MR in the MR-Cons category is chosen, in follow-up execution, only s_0 would be transformed to new s_1 and D_1 would be equal to D_0 . Only if any MR in the MR-All category

is chosen, both s_0 and D_0 would be transformed to new s_1 and D_1 . Then, one can accordingly feed D_1 and s_1 to the CC implementation and let the implementation execute again (as the follow-up execution shows in Fig. 3). After that, one can compare the obtained source and follow-up outputs together and examine whether the concerned MR is violated or not.

IV. EVALUATION

In this section, we evaluate the performance of our designed metamorphic relations on both buggy implementations constructed by mutation testing and real-life CC implementations.

A. Research Questions

We aim to answer the following two questions:

- **RQ1 (Effectiveness)** How effective are our designed MRs in exposing mutated bugs in CC implementations?
- **RQ2 (Usefulness)** How effective are our designed MRs in exposing real-life bugs in CC implementations?

B. Evaluation Design and Setup

1) *Subjects:* We selected three state-of-the-art CC implementations, namely CMID, MG, and INFUSE, from their websites [31], [32], [39]. We refer to them by Imp1, Imp2, and Imp3. They conduct constraint checking with different efficiency and scalability capabilities and can be configured by four modes. We obtained a total of nine implementation variants for the three subjects (three for Imp1, two for Imp2, and four for Imp3). For example, when using Imp3, we deploy it with different variants, which together cover all Imp3's kernel checking logics. We refer to these four variants as Imp3_1 (fully checking mode), Imp3_2 (incremental checking mode), Imp3_3 (concurrent checking mode), and Imp3_4 (fusion checking mode) in order.

2) *Process:* To evaluate the effectiveness of our designed MRs in exposing bugs, we first prepare source test inputs including both data under checking and a constraint for checking. As aforementioned, we generated well-formed FOL constraints following existing work [10] and thus obtained a total of 1,658 well-formed FOL constraints under the height of no more than four. For each pattern associated with each constraint, we accompany it with randomly synthesized data with one or two elements of random values for each associate pattern. Thus, we can treat each test input with data and a specific constraint as the source input in metamorphic testing. Then, by selecting a certain MR, we can obtain a corresponding follow-up test input according to each MR's descriptions as in Table I. Then, by comparing the source and follow-up outputs between two executions, we can tell whether such MR can reveal any violation. To be more specific, we ran each MR's experiment repeatedly with exhaustively enumerated values for each concerned *bfunc* node in calculation during the whole constraint checking. If an MR reveals any violation, we say an implementation bug is detected.

TABLE II
RESULTS OF TESTING OUR DESIGNED MRS ON MUTATED CC IMPLEMENTATION VARIANTS

Subject	#Mutants	#Killed mutants	Killing rate	Number of mutants killed by our designed MRs															
				MR1-1*	MR1-2	MR2-1	MR2-2	MR2-3	MR2-4	MR2-5	MR2-6	MR2-7	MR2-8	MR2-9	MR3-1	MR3-2	MR4-1	MR4-2	
Imp3_1	118	112	94.9%	16	48	48	51	49	49	46	47	45	45	32	16	57	54	59	
Imp3_2	305	256	83.9%	55	103	76	78	97	97	97	101	77	83	81	52	171	102	156	
Imp3_3	161	149	92.6%	36	67	65	72	61	62	61	61	61	61	46	35	86	82	87	
Imp3_4	711	509	71.6%	173	255	180	174	132	128	142	158	127	147	155	172	353	293	318	
Total	1,295	1,026	79.2%	280	473	369	375	339	336	346	367	310	336	314	275	667	531	620	

* To save space, we use MR1/MR2/MR3/MR4-x to represent MR-Data/MR-Cons/MR-All/MR-CCT x in evaluation.

TABLE III
EFFECTIVENESS COMPARISONS UNDER DIFFERENT CONSTRAINT HEIGHTS

Subject	#Mutants	Constraints' height ≤ 3		Constraints' height = 4	
		#Killed mutants	Killing rate	#Killed mutants	Killing rate
Imp3_1	18	18	100.0%	18	100.0%
Imp3_2	13	11	84.6%	12	92.36%
Imp3_3	13	13	100.0%	13	100.0%
Imp3_4	7	4	57.1%	6	85.7%
Total	51	46	90.2%	49	96.1%

3) *To answer: To answer RQ1*, we selected Imp3 as it is the latest CC implementation we know, and used PIT [40], a state-of-the-art mutant generator to randomly generate mutants for Imp3. After manually filtering out equivalent mutants (with simply equivalent code conversions) and low-quality ones (mutating only some command-line processing codes rather than CC kernels), we finally obtained 1,295 CC mutants in total. We categorize each mutant by comparing the mutated codes to logics invoked by Imp3's different variants, and thus obtained 118 mutants for Imp3_1, 305 mutants for Imp3_2, 161 mutants for Imp3_3, and 711 mutants for Imp3_4. Then, by repeating our experiment process to run both source and follow-up test inputs prepared for each MR, we can measure how many violations have ever been observed for each mutant. If any violation is observed, we call it as this mutant successfully killed by the MR. **To answer RQ2**, we evaluate how effective our designed MRs can help find real-life bugs in CC implementations. We selected all our obtained real CC implementations, Imp1, Imp2, Imp3 along with all their variants. Following our experimental process, we evaluate how our designed MRs can reveal violations with the data and constraints prepared as aforementioned. We thus evaluate how our designed MRs reveal violations in real CC implementations, a.k.a., detecting real bugs. We would also investigate the root causes behind MR violations and report our detected bugs for developers' confirm/fixing.

4) *Configuration*: All experiments were conducted on a server with one 48-core Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz and 128GB RAM, installed with Ubuntu 18.04.6 LTS, Python 3.6 and OpenJDK 17.0.7.

C. Result Analyses

1) *RQ1(Effectiveness)*: Table II shows the number of mutants killed by each MR for Imp3's four CC implementation variants, i.e., Imp3_1, Imp3_2, Imp3_3, and Imp3_4.

From Table II, we can observe that our designed MRs can successfully kill 94.92%, 83.93%, 92.55%, and 71.59% mutants together for the four studied implementation variants,

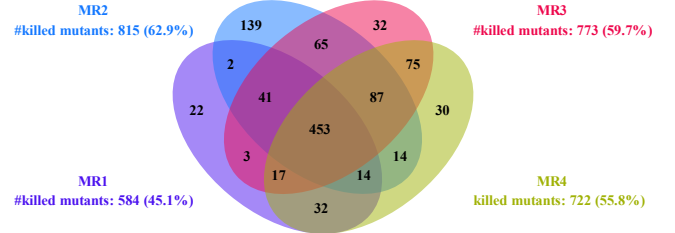


Fig. 5. Venn for the intersections of mutations killed by each MR category

exhibiting their nice effectiveness in revealing violations altogether. After investigating a little deeper into each MR's individual effectiveness, we can further observe that: (1) most MRs show comparable effectiveness in killing mutants with almost comparable number of killed mutants for each implementation variants; (2) MRs can contribute and complement to each other, and together show clearly better effectiveness in killing mutants, e.g., only around 40 killed mutants for each MR v.s. over 100 for all MRs together for Imp3_1.

In order to investigate the effectiveness with respect to different MR categories (i.e., MR1/2/3/4 standing for MR-Data/Cons-/All-/CCT to save space), we accumulate MRs' killing results in each category together as shown in Fig. 5. This venn graph exhibits how different each MR category behaves with respect to the corresponding mutants that have been killed. We can observe that generally, the four categories of MRs exhibit comparable effectiveness with close killing rates around 50%, i.e., 45.1% for MR1, 62.9% for MR2, 59.7% for MR3, and 55.8% for MR4, suggesting that they are similarly valuable in exposing mutated bugs. Among all, MRs in the MR2 category (i.e., MR-Cons) show the best effectiveness on killing the most mutants, i.e., 815 mutants with around 80% being killed among all 1,026 killed ones by all MRs. Still, we also observe that all the four categories of MRs have their unique mutants that can only be killed by themselves respectively, e.g., 22 mutants only killed by MR1, 139 mutants only by MR2, 32 mutants only by MR3, and 30 mutants only by MR4. This suggests that the MRs designed following our principled designing routine have their own values and can be diverse in testing unique functionalities.

Moreover, to make our experiment more cost-effective, we chose to test our subjects using all designed MRs fed by constraints with a height of no more than three only, because experiments on constraints with a height being four or more can be extremely time-consuming and may take even months.

TABLE IV
RESULTS OF TESTING OUR DESIGNED MRs ON REAL-LIFE CC IMPLEMENTATION VARIANTS

CC	Variant	MR1-1	MR1-2	MR2-1	MR2-2	MR2-3	MR2-4	MR2-5	MR2-6	MR2-7	MR2-8	MR2-9	MR3-1	MR3-2	MR4-1	MR4-2
Imp1	_1	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✗	✗	-	-
	_2	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✗	✗	-	-
	_3	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	-	-
Imp2	_1	✓	✓	✓	✓	✗	✗	✓	✓	✗	✗	✗	✓	✓	-	-
	_2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	_3	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Imp3	_1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	_2	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	_3	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗
	_4	✗	✓	✓	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗

“-” represents that in this specific implementation variant, CCT structures cannot be accessible thus MRs in MR4 cannot be used for testing.

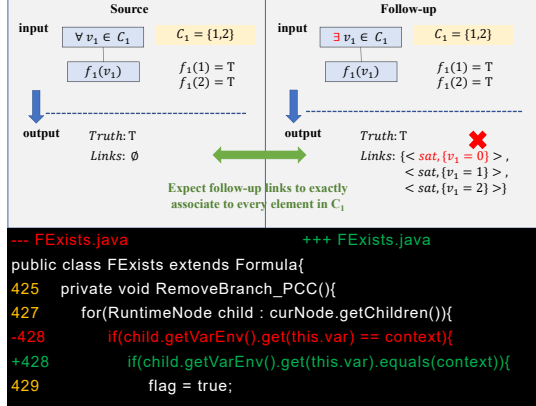


Fig. 6. MR2-1 violation and the fix patch for Imp3_2

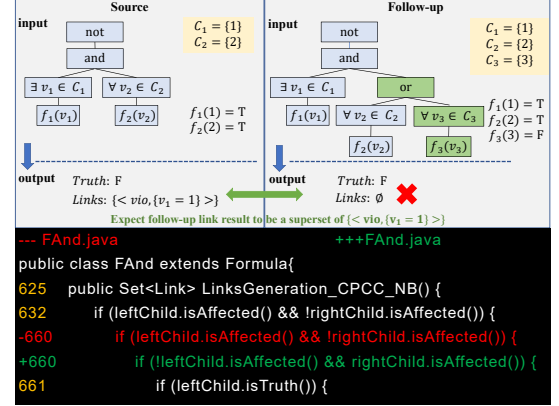


Fig. 7. MR3-1 violation and the fix patch for Imp3_4

Still, to alleviate the possible bias, we randomly sampled 51 mutants (with equivalent and low-quality ones removed) and evaluated the effectiveness of all MRs when feeding constraints with a height of no more than three and of four, as shown in Table III. We observe that 46 mutants have been killed by our MRs for the former, and 49 mutants killed for the latter. Concerning that these three more killed mutants (only 5% increases on the killing rate) come with extremely larger experiment time (over 700x), e.g., around 80 minutes v.s. 1.5 months on average for each mutant, we believe that only using constraints with a height of no more than three is more cost-effective and would not affect our MR effectiveness too much, with an acceptable 5% sacrifice on the killing rate. On the other hand, when selecting MRs in MR2 (the most efficient ones already) by feeding constraints with the height of four, we also observe a similar 5% increase on the killing rate, with a sacrifice of over 250x on the testing time. Therefore, we can safely conclude that using only constraints with a height of no more than three can be quite cost-effective, and sufficient in exposing bugs with a satisfactory killing rate of around 80%.

Therefore, we answer RQ1 as follows: *Our designed MRs can reveal violations effectively to expose mutated bugs with a satisfactory killing rate of around 80%. Different categories of our MRs also show their unique effectiveness in testing.*

2) *RQ2 (Usefulness)*: To evaluate how our designed MRs can expose real-life bugs in CC implementations, we apply them to all the obtained CC implementations. In addition to

the four variants Imp3_1 to Imp3_4 mentioned above, we similarly set the checking option of Imp1 to “fully-checking”, “incremental-checking”, and “concurrent-checking” (Imp1’s all supported options) separately to get Imp1_1 to Imp1_3. And we got Imp2_1 and Imp2_2, with either inner CCT structures accessible or not for Imp2. Each implementation variant represents setting the corresponding implementation with a specific checking mode, thus invoking different kernel checking logics. Table IV shows the results of how our designed MRs reveal violations and detect real bugs for such CC implementation variants. The symbol “✓” means that the corresponding MR does not reveal any violation, while the symbol “✗” means that the MR reveals violations successfully, indicating the existence of implementation bugs.

From Table IV, we observe that: (1) all MRs can reveal violations for at least three real-life implementation variants (at most: five), (2) among all the nine variants, most of them (7/9, 77.8%) have been detected with bugs, and (3) for all the three CC implementations, all of them (3/3, 100%) have been detected with bugs in at least one variant by our MRs. These observations suggest that our designed MRs are useful in detecting real bugs in practical CC implementations. Based on test cases that trigger such MR violations, we can accordingly localize their root causes and successfully detect five unique implementation bugs, which had remained undetected for quite a long time (months to years). After reporting to developers, all of them have been confirmed and fixed.

We illustrate and analyze two of the MR violation cases and their corresponding real bugs next. Fig. 6 shows the MR violation that is revealed when applying MR2-1 on testing Imp3_2. According to MR2-1, if the truth value is T in the source execution, and we transform s_0 's root quantifier from $\forall$ to $\exists$, the follow-up output links should include satisfaction links, each exactly associated to an element in C_1 . However, the follow-up output links accidentally include an unexpected element $\langle \text{vio}, \{v_1 = 0\} \rangle$, which does not associate with any element in C_1 . This violates the assertion required in MR3-1. After digging into the root causes of this violation, we localize an implementation bug, which is caused due to a misuse of “==” instead of “equals” when comparing two Java objects. Therefore, the virtual element 0 added initially only when the set is empty would never be deleted as expected afterward. Fig. 7 shows the MR violation that is revealed when applying MR3-1 on testing Imp3_4. According to MR3-1, if the truth value is F in the source execution, we now transform s_0 by linking a negative quantifier $\forall$ with or. The follow-up output link should contain all the links in the source output. However, the follow-up output link is unexpectedly empty, violating the assertion required in MR3-1. We localized an implementation bug which is caused due to an if-condition being mistakenly written when calculating link explanations for and nodes.

Therefore, we can answer RQ2 as follows: *Our designed MRs can reveal violations as expected and successfully detect real-life bugs in practical CC implementations.*

D. Threat Analyses and Discussion

One potential threat concerns the external validity of our experiments and conclusions since we measure the effectiveness by the killing rate of detecting bugs inserted by mutation testing, thus probably different from real bugs. We argue that mutants have been observed to provide good guidance and showed correlations with real bugs in existing researches [41], [42]. Still, to alleviate the possible bias, we also evaluated our MRs on real-life CC implementations, which also showed positive results in detecting real bugs. Another potential threat may concern our input synthesis for obtaining data under checking and constraints for checking. We synthesize well-formed FOL constraints and accompany each with data containing all associated patterns with simulated elements. Although the process can be simple, we simulate all possible evaluation situations by enumeratedly assigning different *bfunc* values during constraint checking, to make our experiments exhaustive, thus also controlling acceptable testing time as aforementioned.

V. RELATED WORK

Constraint checking is nowadays widely used in maintaining properties for traditional software artifacts, including XML documents [33], [43]–[45], UML models [46]–[49] and distributed source code [50], as well as in emerging dynamic and real-time context-aware software, such as Humanoid Companion Robot [51] and self-driving vehicle systems [52], [53]. Some researchers aim to verify the satisfiability of popular

SAT and SMT expressions and some high-performance solvers (Z3 [54] and CVC4 [55]) have been developed. To check complex properties concerned in software artifacts, some researchers aim to check constraints with richer semantics. Ly et al. [56] emphasizes on compliance monitoring for referencing data beyond the properties in the respective events from the software. Mayr-Dorn et. al [57] proposes a more proactive framework for automatically checking QA constraints depending on process progress. Except for monitoring software traces, some researchers [7]–[10] aim to detect problematic contexts based on FOL-formed consistency constraints.

Metamorphic testing [13], [58], [59] can alleviate the testing oracle problem by specifying and validating program properties as metamorphic relations (MRs). It has exhibited unique usefulness for testing hard-to-test software systems like search implementations [14], SMT solvers [15], C compilers [16], [17], [60], natural language translators [18], [19], [61], autonomous cars [20], [21], web service [62], database [63], DL programs [64], [65], and etc. For example, Winterer et al. [66] and Yao et al. [15] constructed MRs by transforming formulas under solving, and found bugs in mature SMT solvers Z3 [54] and CVC4 [55]. Different from traditional solvers, which check satisfiability and return solutions for SMT formulas, CC implementations studied in this work check dynamic contexts against FOL constraints with extensible *bfunc* parts, thus encoding richer semantics for property monitoring in practical scenarios. We also leverage MT's superiority on testing hard-to-test complex systems in this work and proposed principled metamorphic transformations designed specifically for CC implementations, also confirming MT's unique effectiveness.

VI. CONCLUSION

In this paper, we studied the problem of testing CC implementations in a metamorphic way. We addressed the challenge of automatically generating test inputs and validating test results by a principled metamorphic transformation approach. The approach is built on our sensitivity and diversity principles, and guided the design of a family of 15 MRs. The experiments reported promising results on these MRs' test effectiveness, largely alleviating developers' burden of checking result labeling or cross-implementation validation.

This work also advocates and exemplifies customized design of MRs for specific software systems, rather than purely following existing general-purpose MRs (e.g., fuzzing inputs and predicting output immutability). We look forward to more efforts in this field, and are working along this line for a unified metamorphic framework.

ACKNOWLEDGMENT

This work was supported by the Natural Science Foundation of China (Grants: 61932021, 62072225) and Jiangsu (Grants: BK20202001, BK20220771). The authors would like to thank the support from the Fundamental Research Funds for the Central Universities of China (020214380102, 020214912220), and Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

REFERENCES

- [1] M. Tubaishat and S. Madria, "Sensor networks: an overview," *IEEE potentials*, vol. 22, no. 2, pp. 20–23, 2003.
- [2] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "A survey on sensor networks," *IEEE Communications magazine*, vol. 40, no. 8, pp. 102–114, 2002.
- [3] C.-Y. Chong and S. P. Kumar, "Sensor networks: evolution, opportunities, and challenges," *Proceedings of the IEEE*, vol. 91, no. 8, pp. 1247–1256, 2003.
- [4] K. Rose, S. Eldridge, and L. Chapin, "The internet of things: An overview," *The internet society (ISOC)*, vol. 80, pp. 1–50, 2015.
- [5] S. Li, L. D. Xu, and S. Zhao, "The internet of things: a survey," *Information systems frontiers*, vol. 17, pp. 243–259, 2015.
- [6] K. N. Qureshi and A. H. Abdullah, "A survey on intelligent transportation systems," *Middle-East Journal of Scientific Research*, vol. 15, no. 5, pp. 629–642, 2013.
- [7] H. Wang, C. Xu, B. Guo, X. Ma, and J. Lu, "Generic adaptive scheduling for efficient context inconsistency detection," *IEEE Transactions on Software Engineering (TSE)*, vol. 47, no. 03, pp. 464–497, mar 2021.
- [8] C. Xu, S.-C. Cheung, W.-K. Chan, and C. Ye, "Partial constraint checking for context consistency in pervasive computing," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 19, no. 3, pp. 1–61, 2010.
- [9] L. Zhang, H. Wang, C. Xu, and P. Yu, "INFUSE: Towards efficient context consistency by incremental-concurrent check fusion," in *Proceedings of the 38th International Conference on Software Maintenance and Evolution (ICSME 2022)*, Limassol, Cyprus, Oct. 2022, forthcoming.
- [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 38th IEEE International Symposium on Software Reliability Engineering (ISSRE 2022)*, Charlotte, North Carolina, USA, Oct. 2022, pp. 13–24.
- [11] Earl, T., Barr, Mark, Harman, Phil, McMinn, Muzammil, Shahbaz, and Shin, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering (TSE)*, vol. 41, no. 5, pp. 507–525, 2015.
- [12] L. Filippini, A. Vitaletti, G. Landi, V. Memeo, G. Laura, and P. Pucci, "Smart city: An event driven architecture for monitoring public spaces with heterogeneous sensors," in *2010 Fourth International Conference on Sensor Technologies and Applications*, 2010, pp. 281–286.
- [13] T. Y. Chen, "Metamorphic testing: A simple approach to alleviate the oracle problem," in *Fifth IEEE International Symposium on Service Oriented System Engineering*, 2010.
- [14] Q. Z. Zhi, J. Z. Shu, M. Hagenbuchner, T. H. Tse, F. C. Kuo, and T. Y. Chen, "Automated functional testing of online search services," *Software Testing, Verification and Reliability (STVR)*, vol. 22, no. 4, pp. 221–243, 2012.
- [15] P. Yao, H. Huang, W. Tang, Q. Shi, and C. Zhang, "Skeletal approximation enumeration for smt solver testing," in *29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*, 2021.
- [16] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *2014 Programming Language Design and Implementation (PLDI 2014)*, 2014.
- [17] Q. Tao, W. Wu, Z. Chen, and W. Shen, "An automatic testing approach for compiler based on metamorphic testing technique," in *2010 17th Asia Pacific Software Engineering Conference (APSEC 2017)*, 2011.
- [18] S. Chen, S. Jin, and X. Xie, "Testing your question answering software via asking recursively," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE 2021)*. IEEE, 2021, pp. 104–116.
- [19] Q. Shen, J. Chen, J. M. Zhang, H. Wang, S. Liu, and M. Tian, "Natural test generation for precise testing of question answering software," in *2022 37th IEEE/ACM International Conference on Automated Software Engineering (ASE 2022)*. IEEE, 2022.
- [20] A. Dwarakanath, M. Ahuja, S. Sikand, R. M. Rao, R. Bose, N. Dubash, and S. Podder, "Identifying implementation bugs in machine learning based image classifiers using metamorphic testing," in *27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*, 2018, pp. 118–128.
- [21] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, "Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems," in *IEEE/ACM International Conference on Automated Software Engineering*, 2018.
- [22] Y. Yang, Z. Li, H. Wang, C. Xu, and X. Ma, "Towards effective metamorphic testing by algorithm stability for linear classification programs," *Journal of Systems and Software (JSS)*, vol. 180, no. 5, p. 111012, 2021.
- [23] Y. P. C. S. Xie, Xiaoyuan, "Boosting the revealing of detected violations in deep learning testing: A diversity-guided method," in *2022 37th IEEE/ACM International Conference on Automated Software Engineering (ASE 2022)*. IEEE, 2022.
- [24] T. Chen, D. Huang, T. Tse, and Z. Zhou, "Case studies on the selection of useful relations in metamorphic testing," in *2004 Ibero-American Symposium on Software Engineering and Knowledge Engineering*, 2004, pp. 569–583.
- [25] T. Chen, F. Kuo, Z. Zhou, and T. Tse, "Metamorphic testing and beyond," in *2003 International Workshop on Software Technology and Engineering Practice*, 2004, pp. 19–21.
- [26] Artifact and docs for principled metamorphic relations. [Online]. Available: <https://zenodo.org/records/10052934>
- [27] C. Xu, S.-C. Cheung, W.-K. Chan, and C. Ye, "Heuristics-based strategies for resolving context inconsistencies in pervasive computing applications," in *2008 The 28th International Conference on Distributed Computing Systems (ICDCS 2008)*. IEEE, 2008, pp. 713–721.
- [28] L. Capra, W. Emmerich, and C. Mascolo, "CARISMA: Context-aware reflective middleware system for mobile applications," *IEEE Transactions on software engineering (TSE)*, vol. 29, no. 10, pp. 929–945, 2003.
- [29] K. T. Patil, V. Bansal, V. Dhateria, and S. K. Narayankhedkar, "Probable causes of RFID tag read unreliability in supermarkets and proposed solutions," in *2015 International Conference on Information Processing (ICIP)*. IEEE, 2015, pp. 392–397.
- [30] S. R. Jeffery, M. Garofalakis, and M. J. Franklin, "Adaptive cleaning for RFID data streams," in *Proceedings of the 32nd international conference on Very large data bases (VLDB 2006)*, vol. 6. Citeseer, 2006, pp. 163–174.
- [31] CMID. [Online]. Available: <https://github.com/growinware-cn/CMID>
- [32] INFUSE. [Online]. Available: <https://github.com/yuzi-zly/INFUSE>
- [33] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein, "xlinkit: A consistency checking and smart link generation service," *ACM Transactions on Internet Technology (TOIT)*, vol. 2, no. 2, pp. 151–185, 2002.
- [34] C. Xu, S.-C. Cheung, and W.-K. Chan, "Incremental consistency checking for pervasive context," in *Proceedings of the 28th international conference on Software engineering (ICSE 2006)*, 2006, pp. 292–301.
- [35] C. Xu, Y. P. Liu, S.-C. Cheung, C. Cao, and J. Lu, "Towards context consistency by concurrent checking for internetware applications," *Science China Information Sciences (SCIS)*, vol. 56, no. 8, pp. 1–20, 2013.
- [36] C. Xu, S.-C. Cheung, and W.-K. Chan, "Goal-directed context validation for adaptive ubiquitous systems," in *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*. Minneapolis, Minnesota, USA: IEEE Computer Society, 2007, pp. 1–10.
- [37] C. Xu, S.-C. Cheung, C. Lo, K.-C. Leung, and J. Wei, "Cabot: On the ontology for the middleware support of context-aware pervasive applications," in *IFIP International Conference on Network and Parallel Computing*. Springer, 2004, pp. 568–575.
- [38] J. Sui, C. Xu, S.-C. Cheung, W. Xi, Y. Jiang, C. Cao, X. Ma, and J. Lu, "Hybrid cpu-gpu constraint checking: Towards efficient context consistency," *Information and Software Technology (IST)*, vol. 74, pp. 230–242, 2016.
- [39] MG. [Online]. Available: <https://github.com/cychen2021/stvr23>
- [40] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "PIT: a practical mutation testing tool for java (demo)," *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016.
- [41] M. Papadakis, D. Shin, S. Yoo, and D.-H. Bae, "Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults," in *Proceedings of the 40th International Conference on Software Engineering (ICSE 2018)*. New York, NY, USA: Association for Computing Machinery, 2018, p. 537–548.
- [42] T. T. Chekam, M. Papadakis, Y. L. Traon, and M. Harman, "An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption," in *Proceedings of the 39th International Conference on Software Engineering (ICSE 2017)*. IEEE Press, 2017, p. 597–608.

- [43] C. Nentwich, W. Emmerich, A. Finkelsteini, and E. Ellmer, "Flexible consistency checking," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 12, no. 1, pp. 28–63, 2003.
- [44] S. P. Reiss, "Incremental maintenance of software artifacts," *IEEE Transactions on Software Engineering (TSE)*, vol. 32, no. 9, pp. 682–697, 2006.
- [45] H. A. Handley, W. Khallouli, J. Huang, W. Edmonson, and N. Kibret, "Maintaining the consistency of sysml model exports to xml metadata interchange (xmi)," in *2021 IEEE International Systems Conference (SysCon)*. IEEE, 2021, pp. 1–8.
- [46] A. Egyed, "Instant consistency checking for the UML," in *Proceedings of the 28th International Conference on Software Engineering (ICSE 2006)*, 2006, pp. 381–390.
- [47] R. S. Bashir, S. P. Lee, S. U. R. Khan, V. Chang, and S. Farid, "Uml models consistency management: Guidelines for software quality manager," *International Journal of Information Management*, vol. 36, no. 6, pp. 883–899, 2016.
- [48] M. Nabil, A. Chaoui, and M. Bettaz, "An approach to uml consistency checking based on compositional semantics," *International Journal of Embedded and Real-Time Communication Systems (IJERTCS)*, vol. 8, no. 2, pp. 1–23, 2017.
- [49] B. Wei and J. Sun, "Leveraging sparql queries for uml consistency checking," *International Journal of Software Engineering and Knowledge Engineering*, vol. 31, no. 04, pp. 635–654, 2021.
- [50] A. Demuth, M. Riedl-Ehrenleitner, and A. Egyed, "Efficient detection of inconsistencies in a multi-developer engineering environment," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. IEEE, 2016, pp. 590–601.
- [51] P.-H. Kuo, S.-T. Lin, J. Hu, and C.-J. Huang, "Multi-sensor context-aware based chatbot model: An application of humanoid companion robot," *Sensors*, vol. 21, no. 15, p. 5132, 2021.
- [52] (2020) California DMV. 2020. autonomous vehicle disengagement reports. <https://www.dmv.ca.gov/portal/file/2020-autonomous-vehicle-disengagement-reports-csv/>. [Online]. Available: <https://www.dmv.ca.gov/portal/file/2020-autonomous-vehicle-disengagement-reports-csv/>
- [53] The numbers dont lie: Self-driving cars are getting good. [Online]. Available: <https://www.wired.com/2017/02/california-dmv-autonomous-car-disengagement/>
- [54] L. D. Moura and N. Björner, "Z3: An efficient smt solver," pp. 337–340, 2008.
- [55] M. D. L. H. D. J. T. K. A. R. Clark Barrett, Christopher Conway and C. Tinelli, "CVC4," p. 171–177, 2011.
- [56] L. T. Ly, F. M. Maggi, M. Montali, S. Rinderle-Ma, and W. M. P. Van, der Aalst, "Compliance monitoring in business processes: Functionalities, application, and tool-support," *Information Systems*, vol. 54, no. 1, pp. 209–234, 2015.
- [57] C. Mayr-Dorn, M. Vierhauser, S. Bichler, F. Keplinger, J. Cleland-Huang, A. Egyed, and T. Mehofer, "Supporting Quality Assurance with Automated Process-Centric Quality Constraints Checking," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE 2021)*. Madrid, ES: IEEE, May 2021, pp. 1298–1310.
- [58] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés, "A survey on metamorphic testing," *IEEE Transactions on software engineering (TSE)*, vol. 42, no. 9, pp. 805–824, 2016.
- [59] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. Tse, and Z. Q. Zhou, "Metamorphic testing: A review of challenges and opportunities," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 1–27, 2018.
- [60] 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*, 2016, pp. 849–863.
- [61] S. Chen, S. Jin, and X. Xie, "Validation on machine reading comprehension software without annotated labels: A property-based method," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*, 2021, pp. 590–602.
- [62] W. Chan, S. C. Cheung, and K. R. Leung, "Towards a metamorphic testing methodology for service-oriented software applications," in *Fifth International Conference on Quality Software (QSIC 2005)*. IEEE, 2005, pp. 470–476.
- [63] M. Rigger and Z. Su, "Finding bugs in database systems via query partitioning," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–30, 2020.
- [64] Y. Tian, S. Ma, M. Wen, Y. Liu, S.-C. Cheung, and X. Zhang, "To what extent do dnn-based image classification models make unreliable inferences?" *Empirical Software Engineering (EMSE)*, vol. 26, no. 5, p. 84, 2021.
- [65] C. Tsigkanos, P. Rani, S. Müller, and T. Kehler, "Large language models: The next frontier for variable discovery within metamorphic testing?" in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2023)*. IEEE, 2023, pp. 678–682.
- [66] D. Winterer, C. Zhang, and Z. Su, "Validating smt solvers via semantic fusion," in *Proceedings of the 41st ACM SIGPLAN Conference on programming language design and implementation (PLDI 2020)*, 2020, pp. 718–730.