Received: 9 September 2022 | Revised: 19 November 2022 | Accepted: 18 February 2023

DOI: 10.1002/smr.2555

SPECIAL ISSUE - TECHNOLOGY PAPER

Journal of Software: Evolution and Process WILEY

Model-checking-driven explorative testing of CRDT designs and implementations

Yuqi Zhang | Yu Huang | Hengfeng Wei | Xiaoxing Ma

State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China

Correspondence
Yu Huang, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu 210023, China.
Email: yuhuang@nju.edu.cn

Funding information
National Natural Science Foundation of China, Grant/Award Number: 62025202; Cooperation Fund of Huawei-Nanjing University Next Generation Programming Innovation Lab, Grant/Award Number: YBN2019105178SW38

Abstract

Internet-scale distributed systems often replicate data at multiple geographic locations to provide low latency and high availability, despite node and network failures. According to the CAP theorem, low latency and high availability can only be achieved at the cost of accepting weak consistency. The conflict-free replicated data type (CRDT) is a framework that provides a principled approach to maintaining eventual consistency among data replicas. CRDTs have been notoriously difficult to design and implement correctly. Subtle deep bugs lie in the complex and tedious handling of all possible cases of conflicting data updates. We argue that the CRDT design should be formally specified and model checked, to uncover deep bugs which are beyond human reasoning. The implementation further needs to be systematically tested. On the one hand, the testing needs to inherit the exhaustive nature of the model checking and ensures the coverage of testing. On the other hand, the testing is expected to find coding errors which cannot be detected by design level verification. Toward the challenges above, we propose the model-checking-driven explorative testing (MET) framework. At the design level, MET uses TLA+ to specify and model check CRDT designs. At the implementation level, MET conducts model-checking-driven explorative testing, in the sense that the test cases are automatically generated from the model-checking traces. The system execution is controlled to proceed deterministically, following the model-checking trace. The explorative testing systematically controls and permutes all nondeterministic choices of message reorderings. We apply MET in our practical development of CRDTs. The bugs in both designs and implementations of CRDTs are found. As for bugs which can be found by traditional testing techniques, MET greatly reduces the cost of fixing the bugs. Moreover, MET can find subtle deep bugs which cannot be found by existing techniques at a reasonable cost. Based on our practical use of MET, we discuss how MET provides us with sufficient confidence in the correctness of our CRDT designs and implementations.

Conflict-free replicated data type (CRDT) is a framework that provides a principled approach to maintaining eventual consistency among data replicas in distributed systems. CRDTs have been notoriously difficult to design and implement correctly. We propose model-checking-driven explorative testing (MET) framework for dealing with such problem. We apply MET in our practical development of CRDTs. MET successfully finds subtle deep bugs and provides us with sufficient confidence in the correctness of our CRDT designs and implementations.

KEYWORDS
CRDT, explorative testing, model checking, TLA+

1 | INTRODUCTION

Large-scale distributed systems often resort to replication techniques to achieve fault-tolerance and load distribution.[1–3] For a large class of applications, user-perceived latency and overall service availability are widely regarded as the most critical factors. Thus, many distributed systems are designed for low latency and high availability in the first place and resort to *eventual consistency* due to the CAP theorem.[4,5] Eventual consistency allows replicas of some data type to temporarily diverge and makes sure that these replicas will eventually converge to the same state in a deterministic way.[2] The conflict-free replicated data type (CRDT) framework provides a principled approach to maintaining eventual consistency.[1,2] CRDTs are key components in modern geo-replicated systems, such as Riak,[6] Redis-Enterprise,[7] and Cosmos DB.[8]

It has been notoriously difficult to correctly design and implement CRDTs. CRDT implementations suffer from so-called *deep bugs*.[9]

From the extensional perspective, deep bugs often appear in rare situations, but they still manifest themselves, often in critical situations, in large or longtime deployments.

From the intensional perspective, due to the intrinsic uncertainty in distributed system execution, deep bugs only appear when the system is faced with certain subtle combinations of system and environment events. Though the bug-triggering patterns of events usually involve only a moderate number of events, the space of all potential bug-triggering patterns is exponentially large. This explains why deep bugs are hard to replay and why they are often beyond the coverage of standard testing techniques.

It is a great challenge to prevent, detect and fix deep bugs in the design of a CRDT. The central issue in CRDT design is to resolve conflicts between concurrent updates, no matter how the updates are out of order on different replicas. Design of the conflict resolution strategy is typically tedious and error prone, especially for data types with complex semantics. The designer has to exhaustively check all possible interleavings of conflicting updates, which results in a great amount of metadata. The metadata needs continuous and consistent maintenance. Existing CRDT designs are mainly informal. The correctness of the design highly depends on the experiences of the designer. The design also needs intensive design review. All these factors make the design process quite time and energy consuming.

In order to cope with subtle deep bugs in the design of a CRDT, it is necessary to have a precise and unambiguous description of the design. More importantly, the design should be "debuggable." That is, the design needs to be automatically explored by a machine. All corner cases can be covered by exhaustive exploration. Moreover, when a bug is found in the design, the exploration process is also expected to provide sufficient information for the designer to find the root cause of the bug and then fix it.

It is also a great challenge to cope with deep bugs in the implementation of a CRDT, even after the design is deemed correct by formal verification.

First, the implementation has to handle details which are not covered in the design. The transcription from the design to the implementation often introduces subtle bugs. Moreover, the design is intended for human reasoning. The developer may often find the design inefficient when directly translated to the implementation. Thus the developer often has a strong incentive to optimize the design. Such intentional deviations often increase the odds of transcription errors in the implementation.

Second, the design often makes "reasonable" assumptions. In CRDT implementations, such assumptions have to be carefully ensured by extra implementations which are neglected in the design. The implementations centering around assumptions in the design are expected to be straightforward and highly dependable. However, this is often not the case when implementing complex CRDTs. For example, ensuring an assumption in the design often makes use of third-party libraries. Such libraries often have many options for different usage in different scenarios. Misconfiguration or misuse of the libraries may introduce subtle bugs. Such bugs often evade human reasoning and code review because the corresponding implementations are not specified in detail in the design.

Existing testing techniques are insufficient to cope with deep bugs in CRDT implementations. Formal specification and verification tools such as TLA+ suits are target at checking the distributed protocols. They can not be directly applied at CRDT implementations. Code-level model checking for distributed systems can handle deep bugs, but usually imposes a prohibitive cost. What we need is to achieve the best of both worlds. We need the ability of distributed system model checking to exhaustively check all corner cases of system execution, but also need the cost-effectiveness of standard testing techniques.

Toward the challenges above, we propose the model-checking-driven explorative testing (MET) framework. As indicated by its name, the MET framework consists of two layers. In the *design layer* (also denoted the *model layer*), MET employs the Temporal Logic of Actions (TLA+) for the specification of CRDT protocols.

TLA+ is a lightweight formal specification language especially suitable for the design of distributed and concurrent systems.[10] The specification in TLA+ precisely describes what is required (i.e., specification of the correctness properties) and what is to be implemented (i.e., specification of the CRDT protocol).

More importantly, the design specified in TLA+ is model checked to eliminate any violation of the user-specified correctness conditions. The key to practical and effective model checking is to tame the state explosion problem. MET coarsens the specification of the CRDT protocol to prune unnecessary interleavings of events. MET also directly limits the scale of the system model to reduce the model-checking cost. MET leverages the designer's understanding of the CRDT protocol to ensure that the pruning of the state space will not significantly hamper the ability of MET to "dig out" deep bugs.

In the *implementation layer* (also denoted the *code layer*), MET conducts *explorative testing* on CRDT implementations, in the sense that the testing systematically controls and permutes all nondeterministic choices of message reorderings.

MET inherits the exhaustive exploration from the design layer to the implementation layer. The explorative testing contains two key components: generation of test cases and enhancement of system testability.

As for test case generation, the test cases are automatically extracted from the model-checking trace. The model-checking trace consists of a sequence of system states connected by events triggering the state transitions. The sequence of events, denoted as the event schedule, is extracted as the test input. The events mainly include system events, for example, client requests, synchronization among replicas, and environment events, for example, (out-of-order) message delivery.

The sequence of system states is extracted as the test oracle. That is, the code-level execution is expected to produce the same sequence of system states as the model-level execution.

As for enhancement of system testability, MET first enhances system controllability. That is, MET deterministically replays the model-checking trace in the implementation layer. This is achieved by hacking the underlying RPC among server replicas of the CRDT store. The communications among server replicas are all directed to a central test manager. The test manager discharges the intercepted events one by one, strictly following the event schedule in the test input.

Note that this hacking is transparent to the upper layer CRDT implementation.

Besides system controllability, MET further needs to enhance system observability. MET does not adopt the typical approach, where the system states are logged and then analyzed offline after the test case execution. MET first implements dedicated APIs for the test manager to inspect the internal states of the server replicas. Given the controllability of the system, the test manager intercepts all system events and decides the total order of the events. Then the test manager inserts the system state inspection commands into the event schedule to record the system state. This approach greatly simplifies the work of comparing the state sequence of system execution with that of the model checking, i.e., the test oracle.

The MET framework is applied in the design and implementation of different types of replicated priority queues (RPQs) and replicated lists (RLists) over the CRDT-Redis data type store.[11] We first discuss how our experiences in testing CRDT implementations motivate the MET framework. Second, we discuss the bugs found by MET in both design and implementation of CRDTs. The MET framework greatly eases the fixing of bugs which can be detected by standard testing techniques. MET also finds bugs which cannot be detected by standard testing techniques. Third, we discuss how the exhaustive (constrained by the testing budget) and explorative testing of CRDTs using MET increases our confidence in the correctness of the design and implementation of CRDTs.

The rest of this work is organized as follows. Section 2 extensively discusses the motivation for MET and overviews the design of MET. Sections 3 and 4 present the model layer and the code layer design of MET, respectively. Section 5 demonstrates the application of MET in practical CRDT designs and implementations. Section 6 discusses the related work. In Section 7, we conclude this work and discuss the future work.

2 | MOTIVATION AND OVERVIEW

In this section, we first present our practical experiences in CRDT development to extensively explain the motivation behind MET. Then, we provide an overview of MET.

2.1 | Our experiences motivating MET

The MET framework is proposed in our practices in developing different types of CRDTs. Specifically, we propose the RWF conflict resolution strategy to ease the design and implementation of data container CRDTs. RWF-RPQ and RWF-List are developed on the CRDT-Redis data type store.[11] For the purpose of performance comparison, we also develop the RemoveWin-RPQ and RemoveWin-List. RWF and RemoveWin can be viewed as two different types of conflict resolution strategies. The details of these strategies are irrelevant to our discussions on the MET framework here. More details of these CRDTs can be found in Zhang et al.[12]

The key challenge in CRDT development is to ensure that the conflict resolution strategy guarantees eventual consistency, no matter how the data updates are out of order. Lacking an explorative testing service or tool, we resort to random stress testing. We keep generating

concurrent and conflicting data updates and randomly dispatch the updates to all the replicas. We further use traffic control (TC)[13] to add random delay to the messages among server replicas to make the updates out of order.

The testing proceeds in rounds. In each round, 10,000 update operations are generated per second, lasting 5 min. We check whether the replicas reach the same state after they have received all the data updates at the end of each round. A bug is found if any two replicas do not reach the same state. If no bug is found after 24 h of stress testing, the implementation is deemed (sufficiently) correct. More details of the CRDT development can be found in our previous work.[12]

The experiences in testing our CRDT implementations extensively motivate us to design the MET framework, as detailed below.

2.1.1 | Why we need debuggable CRDT design

The conflict resolution logic in a CRDT protocol is usually quite complex and error prone. It is quite difficult to ensure the correctness of a CRDT protocol only based on the informal design.

However, informal software designs, for example, textual descriptions, flow charts, and pseudocodes, are dominant in current software development practices. The correctness of informal software design is mainly guaranteed by manual reasoning and intensive design review. It is widely accepted that human intuition is poor at estimating the true probability of supposedly extremely rare combinations of events in real deployments of complex distributed systems.[9,14]

For example when developing the RWF-List,[12] we find violations of eventual consistency through stress testing in a small number of experiments, as shown in Issue #1 below.

Issue #1. List replicas may not converge in the stress testing.[15]

To find the root cause of this bug, we first look at the final states of two replicas and identify the list elements with divergent positions. As specified in the RWF-List protocol, the position of one element is decided when the element is added to the list for the first time. Thus, we go backtracking through the execution trace, in order to find the point when the list replicas first diverge, and the point when the elements with divergent positions are first added to the list. In this way, we find that the bug is caused by the element which is in the list but has an undefined position.

A minimized example of this bug is shown in Figure 1. According to our design of the RWF-List, the location of list element e is decided when it is first added on $server_0$. If we remove and then re-add e (such as undoing the removal of words in collaborative editing), its position will not change. The client may send the *re-add*(e) operation to $server_1$, which has not yet gotten the previous *add*(e) operation from $server_0$. $Server_1$ should reject this *re-add* operation or buffer this operation for processing later. But in our buggy design, $server_1$ accepts this request and assigns an illegal position to e. When $server_1$ receives the delayed *add*(e) operation, it will ignore this operation, which causes the divergence among replicas.

Though the debugging process above is methodologically simple, it is quite time and energy consuming. The stress testing runs for hours or even days to find the bug. The error trace is quite long when the divergence among replicas is identified. Both trace analysis scrips and manual inspection are needed to find when the divergence first occurred and when the elements corresponding to the divergence were first added to the list. Given the information about the divergence, we then need to manually reason how the elements are handled by the replicas, in order to understand how the bug is introduced in the design and the implementation. We spent several days to find the root cause and fix this bug.

Note that in our design, the *re-add* operation is treated as a special case of the *add* operation. To fix the bug in Issue #1, we add one more case in the conflict resolution logic of the *add* operation. The resulted implementation then passes the stress testing. However, we highly suspect that there are still deep bugs hidden in our design. It is mainly because the conflict resolution logic in the *add* operation is quite complex. Further adding more cases in the already-quite-complex logic in *add* makes the design highly unreliable. Manual reasoning is powerless to ensure the correctness of this complex design.

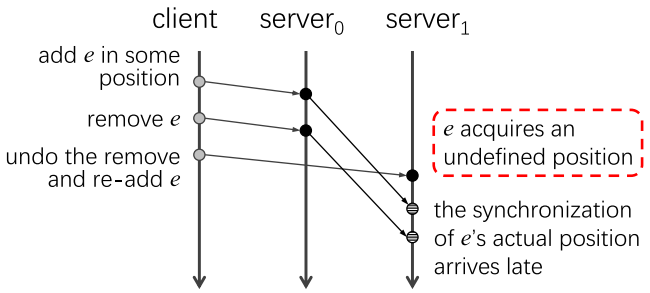

FIGURE 1 Buggy design of the re-add operation.

The experiences above convince us to add formal specification and verification as one obligatory step in our CRDT design. Though formal specification also requires nontrivial human efforts, the specification process will force the designer to be unambiguous about the details in the design. The specification can further be automatically and exhaustively checked by a model checker. We expect the increase in design quality and decrease in testing cost can well compensate for the human efforts in the formal specification process.

2.1.2 | Why we need code-level testing

Even though the design has been deemed correct after the formal verification, we still need code-level testing. It is mainly because, at the model level, we verify a system model, not the actual system implementation. The obtained results are just as good as the system model. Different types of coding errors can be introduced in the implementation. We mainly discuss two salient types of coding errors, namely the *transcription error* and the *assumption error*.

We first discuss the transcription errors, i.e., errors introduced when transcribing the design into the code. As we know, software design is intentionally abstract. The developer must instantiate the abstract design and transform it to correct executable codes. The developer will meet numerous details which are not covered in the design. This process is often error prone. Moreover, the developer often has the incentive to optimize the design during the implementation process. This is mainly because the informal design is intended for human reasoning. Thus directly translating the design to executable codes is often a feasible but inefficient choice. Even for the formal design intended for machine exploration, there is still a significant gap in semantics between a formal specification and an executable program. Thus the programmer may often try to find an equivalent but more efficient implementation from the abstract design. This process often increases the odds of introducing transcription errors.

Though our random stress testing did not find bugs due to transcription errors (denoted the *transcription bugs*), we highly suspect that there must be such bugs in our implementation. We "borrow" a transcription bug found by the MET framework from Section 5.3.1 for the purpose of illustration here.

Issue #4. In CRDTs developed using the Rwf framework, the existence of one element is protected by a precondition in the design. However, the implementation deviates from the design in the case when the precondition does not hold.[16]

Issue #4 can be illustrated by the example in Figure 2. In this example, $client_0$ first adds element a and then changes the value of a (say, it changes the font of character a in a string in a collaborative editing scenario). Then, $client_1$ queries the value of the string and obtains string "$[a]$." After reading the value of the string, $client_1$ inserts b after a into the string. However, in our buggy implementation, $client_0$ may read string "$[b,a]$."

This bug can be triggered by the following adversarial pattern of events.

$Server_0$ first receives the operation "changing the font of a" (without adding a first). Then, $server_0$ will receive the operation "adding b after a." At the design level, this "adding b after a" operation is quite safe, because it is protected by the precondition that "before adding b after a, a must exist." The design does not explicitly state what to do if the precondition does not hold. The precondition implicitly states that, if the element does not exist, nothing should be done. However, in our implementation, an illegal position is assigned to b and then to a. In this way, $client_0$ gets "$[b,a]$."

This bug is beyond the coverage of model checking in the design layer, and we discuss in Section 5.3.1 how we detect this bug with the help of MET.

The second type of coding errors, namely, the assumption error, pertains to assumptions in the design. In a broader sense, the assumption error is also a kind of transcription error. However, as various assumptions are widely used in designs of distributed protocols and systems, we explicitly separate this type of errors out.

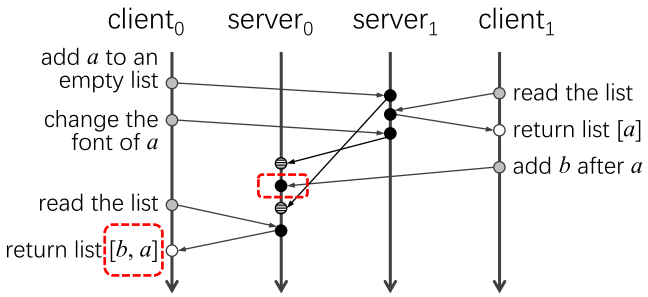

FIGURE 2 An exemplar transcription bug.

In designs of distributed protocols and systems, many details are often intentionally omitted, via the form of "reasonable" assumptions. The assumptions are usually assumed to be straightforward to implement or to be readily provided by existing libraries/tools. Thus details of how the assumptions should be guaranteed are often omitted in the design. However, when different modules of a system have different or even conflicting assumptions, the developer needs to put more effort into managing the implementations pertaining to a number of various assumptions. Moreover, third-party libraries or tools can have complex semantics and subtle configurations. Even if the library implementation is perfectly correct, the developer may still use it with a wrong configuration, thus failing to provide the intended guarantee required by the assumption. In such situations, the developers may introduce assumption errors unwittingly. As the number of assumptions in the design increases, the odds of *assumption bugs*, i.e., bugs due to the assumption errors, quickly increase to an extent that cannot be neglected.

Issue #2 is an example of the assumption bug in our implementation. The RemoveWin CRDTs assume that the underlying network provides the causal delivery semantics, but the networking primitives of the CRDT-Redis data store do not provide such semantics. In contrast, the RWF CRDTs do not need the causal delivery network.

In the example shown in Figure 3, the *client* sends requests a, b, and c to different servers. $Server_1$ receives b and c directly from the *client*, and receives the synchronization of a from $server_0$. Because the synchronization of a arrives at $server_1$ before that of b and c, $server_1$ works correctly. In contrast, $server_2$ gets b and c from $server_1$ first, without getting a. The upper layer RemoveWin CRDT protocol assumes that the underlying network provides causal delivery of messages. Thus the CRDT protocol does not consider the case in our example and b and c are erroneously processed on $server_2$.

The root cause behind this bug is that the RWF CRDTs and the RemoveWin CRDTs have different assumptions on the underlying network. The CRDT-Redis platform aims to host different CRDT protocols, which may have different assumptions on the network, the persistent storage, etc. Thus we need to "align" the assumptions of different protocols on the CRDT-Redis platform.

This situation is similar to the misconfiguration problem in cloud and data center platforms.[17]

The implementations pertaining to such alignments are error prone and are not sufficiently documented in the design.

Also note that, even when we find the root cause of an assumption bug, the fixing of this type of bugs is often nontrivial. In our first patch to fix Issue #2, $server_2$ does not synchronize b and c correctly when it receives a. Then, when $server_2$ later receives d, it gets stuck because the required delivery of b and c are not available. We argue that not only the original implementation, but also the patches fixing the assumption bugs need extensive testing.

Issue #2. The RemoveWin CRDT protocols assume that the underlying network provides causal delivery of messages. The CRDT-Redis platform does not provide such semantics of network communication.[18]

Besides the assumptions concerning the underlying network, the CRDT protocols may also have assumptions about other modules in the data store. More assumption bugs will be discussed in Section 5.3.2.

2.1.3 | Why we need automatic explorative testing

Issue #2 discussed above is discovered by manual explorative testing. Given the effectiveness of this manual explorative testing, we decide to automate this process. This automation gives us the MET framework.

Specifically, we, as the designer of the CRDT protocol, know which part of the conflict resolution is most tricky and unreliable. Thus, we manually construct adversarial executions which target at the most tricky part of our design.

Given the adversarial executions, we then extend our unit testing to replay such executions for one server replica. The inputs are given to the replica following the adversarial execution. The replica then executes its CRDT protocol. When the replica interacts with the outside world, we

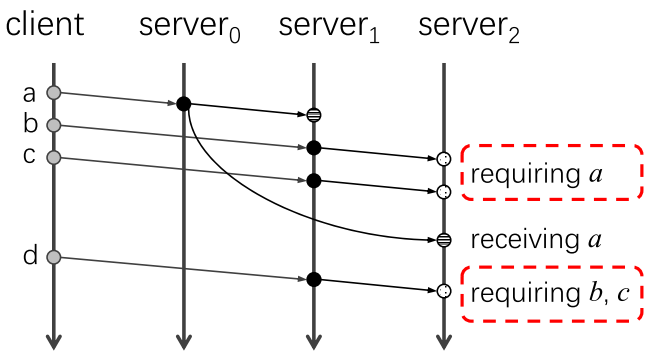

FIGURE 3 An exemplar assumption bug.

manually calculate what should be returned to the replica. The server replica is provided with the illusion that it obtains the feedback from other peer replicas.

Issue #2 is out of reach of stress testing and is found by this manual explorative testing.

We can easily see the advantages and disadvantages of the manual explorative testing.

The main advantage is that the test cases are significantly more effective than the randomly generated ones. The main disadvantage is that the whole process is manual and imposes a prohibitive cost. Our CRDT implementations cannot be sufficiently tested using this manual testing.

The MET framework is directly shaped by this manual explorative testing practice. MET enumerates all possible test cases, thus deterministically covering the most important ones. MET automates the most tedious part of the manual testing and enables efficient explorative testing.

2.2 | Overview of the MET framework

The MET framework consists of two layers. The architecture of MET is illustrated in Figure 4.

In the model layer, we have:

- (Section 3.1) *Specification*. The CRDT protocol and the correctness conditions are formally specified in TLA+.
- (Section 3.2) *Model checking*. The specifications are automatically explored by the TLC model checker. Reduction techniques are proposed to tame the state explosion problem.

In the code layer, we have:

- (Section 4.1) *Test case generation*. Test cases are automatically generated from the model-checking trace. Test case generation includes generation of the *test inputs* and that of the *test oracle*.
- (Section 4.2) *Testability enhancement*. The test manager replays the model-checking trace in the code layer, which requires the enhancement of system *controllability* and *observability*.

3 | FORMAL SPECIFICATION AND MODEL CHECKING OF CRDT DESIGNS

In this section, we introduce the formal verification of CRDT design in MET, including formal modeling in TLA+ and taming of the state explosion problem in model checking.

3.1 | Formal modeling in TLA+

The formal modeling includes the specification of the CRDT protocol and that of the correctness properties. The formal specification language we use is TLA+.

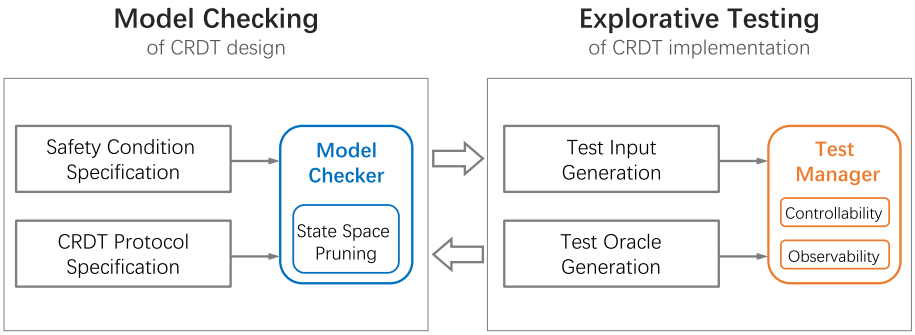

FIGURE 4 Architecture of the MET framework.

3.1.1 | TLA+ Basics

In TLA+, we model a distributed system in terms of one single global state. A distributed system is specified as a state machine by describing the possible initial states and the allowed state transitions called *Next*. The system specification contains a set of (global) system variables *V*. A *state* is an assignment to the system variables. *Next* is the disjunction of a set of actions $a_1 \vee a_2 \vee \ldots \vee a_p$, where an *action* is a conjunction of several clauses $c_1 \wedge c_2 \wedge \ldots \wedge c_q$. A *clause* is either an *enabling condition* or a *next-state update*.

An enabling condition is a state predicate which describes the constraints the current state must satisfy, while the next-state update describes how variables can change in a step (by "step" we mean successive states). Whenever every enabling condition ϕ_a of an action *a* is satisfied in a given "current" state, the system can transfer to the "next" state by executing *a*, assigning to each variable the value specified by *a*. We use "$s_1 \xrightarrow{a} s_2$" to denote that the system state goes from s_1 to s_2 by executing action *a*, and *a* can be omitted if it is obvious from the context. Such execution keeps going and the sequence of system states forms a trace of system execution. Exemplar TLA+ specifications can be found in the following Figures 5 and 6.

One salient feature of TLA+ is that correctness properties and system designs are just steps on a ladder of abstraction, with correctness properties occupying higher levels, system designs and algorithms in the middle, and executable code and hardware at the lower levels.[14] This ladder of abstraction helps designers manage the complexity of real-world distributed systems. Designers may choose to describe the system at several "middle" levels of abstraction, with each lower level serving a different purpose, such as to understand the consequences of finer grain

MODULE *rwf_rpq*

$Init \triangleq$ Global variables
$\wedge\ ops = [j \in Procs \mapsto \{\}]$ network buffer
$\wedge\ history = \langle\rangle$ history variable
Local data and metadata of replicas
$\wedge\ e\_set = [self \in Procs \mapsto \{\}]$
$\wedge\ t\_set = [self \in Procs \mapsto \{\}]$

$Next \triangleq \exists\, self \in Procs :$ choose one replica to act
Generate a client request, handle it
$\vee$ LET $request \triangleq generateClientRequest(\ldots)$ IN
$\wedge\ history' = Append(history, \langle request, \ldots\rangle)$
$\wedge\ \vee\ tryToHandleAsAdd(request, \ldots)$
$\vee\ tryToHandleAsRemove(request, \ldots)$
$\vee\ tryToHandleAsIncrease(request, \ldots)$
Handle a synchronization message
$\vee$ IF $ops[self] \neq \{\}$ THEN $\exists\, msg \in ops[self] :$
$\wedge\ history' = Append(history, \langle msg, \ldots\rangle)$
$\wedge\ syncAndUpdate(msg, \ldots)$

FIGURE 5 A skeletal example of the TLA+ specification for RWF-RPQ.

MODULE *rwf_list*

If two replicas have received the same set of updates, they will converge to the same state with regard to all CRDT data and metadata.
$StrongEventualConsistency \triangleq$
$\forall\, p1, p2 \in Procs : (p1 \neq p2 \wedge ops[p1] = ops[p2])$
$\Rightarrow (e\_set[p1] = e\_set[p2] \wedge t\_set[p1] = t\_set[p2]$
$\wedge\ l\_set[p1] = l\_set[p2] \wedge lt\_set[p1] = lt\_set[p2])$

The elements in the list are always totally ordered, despite that they may be inserted concurrently.
$ListElementTotalOrdered \triangleq$
$\forall\, p \in Procs : \forall\, i, j \in Elmts :$
$(i \neq j) \Rightarrow (l\_set[p][i] = Dft\_leid$
$\vee\ l\_set[p][j] = Dft\_leid$
$\vee\ l\_set[p][i] \neq l\_set[p][j])$

$CheckingInvariant \triangleq StrongEventualConsistency$
$\wedge\ ListElementTotalOrdered$

FIGURE 6 Correctness condition of RWF-List in TLA+.

concurrency or more detailed behavior of a communication medium. The designer can then verify that each level is correct with respect to a higher level. The freedom to choose and adjust levels of abstraction makes TLA+ extremely flexible.

3.1.2 | Specification of CRDT protocols

A replicated data type store consists of a group of replicas and clients, as well as the network channel that connects them. The state of the replica consists of (i) the concrete data of the data type, (ii) the metadata for conflict resolution, and (iii) the input/output message buffer for network communication. The essential issue of specifying a CRDT protocol is to specify how the replica updates its state. The state transfer can be driven by three kinds of events, as defined in the generic framework of CRDT design[2,19]:

- *send*: the *send* event can be triggered when the output buffer of the server replica is not empty. It will broadcast one message in the output buffer to all other replicas.
- *receive*: the *receive* event can be triggered when there are messages to be delivered from the network channel. It will deliver one message from the channel to the input buffer of the server replica.
- *do*: the *do* event can be triggered when the input buffer is not empty. It consumes one message from the input buffer and updates the state of the server replica. If the message is a client request, the *do* event will further put a synchronization message into the output buffer of the replica.

Detailed design of the replica update logic is specified in the CRDT protocol. The event-driven design of a CRDT protocol can be readily transformed to TLA+ specifications. More examples of CRDT protocol specifications can be found in our online repository.[11]

The underlying network is modeled as a shared global channel, described by a global variable in TLA+. The channel buffers all the messages in transmission. We assume that the network channel provides the eventual-delivery-once semantics. The messages can be out-of-order, but cannot be dropped, duplicated, or forged.

We also assume that the replicas will not crash. Note that this assumption is indicated in the definition of eventual consistency[2] and is widely adopted in CRDT designs. Put it in another way, when the replicas can crash or the messages can be dropped, the data type store does not need to guarantee anything and still provides eventual consistency. Thus, such cases are usually not covered in the design of a CRDT protocol.

Given the modeling of the network channel, we do not explicitly model the clients. All client requests are directly modeled as data access messages in the network channel, which will be delivered to the replicas some time in the future.

Note that the specification of system behavior intentionally omits certain details. We will justify these simplifications in Section 3.2.

Figure 5 is a skeletal example of the specification of a CRDT protocol. The global variables record status of the server replica. The *history* variable records the state transitions (the history variable is introduced for the purpose of test case generation, as detailed in Section 4.1.1). The *action* records all possible state transitions. The model checker simply applies all possible state transitions and enumerates all reachable system states.

3.1.3 | Specification of correctness properties

The correctness condition we adopt for CRDT design is eventual consistency.* A CRDT protocol guarantees eventual consistency if any two replicas always converge to the same state as long as they have received the same set of updates, regardless of the order of updates received. The convergence among replicas means that they reach the same state in regard to both the data type itself and the metadata for conflict resolution.

Besides eventual consistency, we may further specify correctness conditions pertaining to the semantics of the data type under concern. For example in the design of the RWF-List, we check whether the position identifiers of all elements in the string are unique and totally ordered.

Figure 6 is an example of specifying the correctness conditions for RWF-List. The correctness condition includes both specification of eventual consistency and that of the constraints on the position identifiers.

3.2 | Model checking of CRDT designs

Given the CRDT protocol specified in TLA+, we can now explore the design using the TLC model checker. Though the model-checking process is fully automatic, it is intrinsically constrained by the state explosion problem. Full checking of the system specification of arbitrary scale is usually impractical.

In order to conduct practical model checking of the design and obtain sufficient confidence in the correctness of the design, we prune the state space in advance, from two orthogonal dimensions. First, the specification is coarsened to omit unnecessary details. Second, we leverage the small scope hypothesis to check a small scale system while not significantly sacrificing the effectiveness of finding deep bugs.

3.2.1 | Coarsening the specification

We utilize the feature that TLA+ enables the developer to flexibly adjust the level of abstraction. In the TLA+ specification of a CRDT protocol, the replica mainly processes three types of events. The *send* and *receive* events model the network communication. The *do* event mainly updates the metadata for conflict resolution and updates the data type itself. In light of the state explosion problem, we coarsen the TLA+ specification and focus on the *do* event.

We bind the *send* event after the *do* event. This means that we let a replica broadcast the synchronization message to all other peer replicas right after it has handled a data update from some client, rather than handle other events first and broadcast the data update in the future. Similarly, we bind the *receive* event before the *do* event. This means that we let a replica handle the update request in a message right after the message is received.

We do not model the library code in detail. We assume that the libraries are implemented correctly and model them as black-box functions according to the specification in their manuals. The inconsistency will be found in code-level testing if such assumption fails.

This coarsening is mainly based on the observation that the subtle and deep bugs in CRDT designs and implementations mainly result from the complex (and often tedious) processing of diverse patterns of conflicting data updates. When we "glue together" the *receive*, *do*, and *send* events, unnecessary interleavings of events are pruned from the state space, while all possible cases of conflicting resolution are preserved. As further evidenced by our experimental evaluation on MET, our coarsening of the specification effectively limits the cost for model checking, while not significantly sacrificing the ability to find deep bugs.

3.2.2 | Limiting the scale of the model

In theory, we need to set the scale of the model to a number larger than any possible instance of the system deployed in a real scenario. Then, the model checking can guarantee that the design is always correct in all possible application scenarios. However, due to the state explosion problem, the checking cost is exponential to the scale of the model. We have to limit the scale to achieve practical checking.

We argue that the small scale model checking can still find most deep bugs. Our argument is backed by the observation known as the *small scope hypothesis*.[20–23] The hypothesis states that analyzing small system instances suffices in practice to ensure sufficient correctness of large scale systems. Empirical studies support this hypothesis in different settings.[21,24,25] For example, in the setting of distributed systems, this hypothesis is proved for a family of consensus algorithms targeting the benign asynchronous setting.[22] An empirical study[21] of 198 bug reports for several popular distributed systems found that 98% of those bugs could be triggered by three or fewer processes.

We leverage our understanding of the CRDT protocol to tune the key parameters of the system configuration in order to control the scale of the model. Let n denote the number of server replicas, q the number of client requests, and Q the number of all possible client requests, that is, the size of the space of all possible requests.

As for the server number n, we set n to a small number. In our application of the MET framework (Section 5), n is up to 3.

In most cases, three replicas are sufficient to manifest the subtle interleavings of concurrent data updates which result in the deep bugs.[26,27] As n goes beyond 3, the checking cost quickly becomes prohibitive.

As for q, we set the $q \geq n$. This is to make sure that every replica is covered. Besides, the client requests also need to cover all APIs the CRDT provides. Given the limit on the testing budget, we can have bigger q when n is small, but when n is relatively large, we can only have q close to or equal to n.

As for Q, we let it be a little larger than q. This enables each client request to possibly have different parameters while limiting the state space. The value of Q is affected by multiple factors. The MET framework is mainly targeted at data container types. We first need to consider whether the data elements in the container are independent. If so, we can consider each element separately, significantly reducing the size of the state space. But we may also face the data types where the data elements cannot be handled separately. For example, elements in the Rwf-RPQ can be viewed as independent,[†] while elements in the Rwf-List are not independent.

When the model checker needs to execute a client request, we set the parameters of the request by choosing a uniformly random request from all possible requests.

When the operations have multiple equally important parameters, we choose one to represent all the parameters.

Detailed configurations of the model and the results of model checking are described in Section 5.4.2.

4 | EXPLORATIVE TESTING OF CRDT IMPLEMENTATIONS

The MET framework provides an explorative testing service to guarantee sufficient correctness of CRDT implementations. The key of the explorative testing service is to utilize the model-checking results at the model level and guide the explorative and exhaustive (constrained by the testing budget) testing at the code level. The two key components of the testing service are automatic *test case generation* from the model-checking trace and *testability enhancement* of the system under test, as detailed below.

4.1 | Test case generation

The salient feature of explorative testing is that it systematically controls and traverses the nondeterministic choices of message reorderings for CRDT implementations. Given that the exhaustive checking of message reorderings has been conducted in the model layer, we automatically generate test cases from the model-checking traces. The test case generation consists of two parts: the generation of test inputs and that of the test oracle, as shown in Figure 7.

4.1.1 | Generation of test inputs

The test input is merely the *event schedule*, that is, a sequence of events that take place in the system. The test input consists of the *system events* and the *environment events*. According to our modeling in Section 3.1, the system events mainly include data update requests from the clients and the data synchronization requests among replicas. The environment events are the delivery of data updates from the network channel to the replicas.

To enable automatic extraction of test inputs from the model-checking trace, we add the history variable[28] in the TLA+ specification of the CRDT protocol. The history variable is an auxiliary variable which passively records the state transitions in system execution without interfering with the execution.

The test inputs can thus be extracted from the history variable.

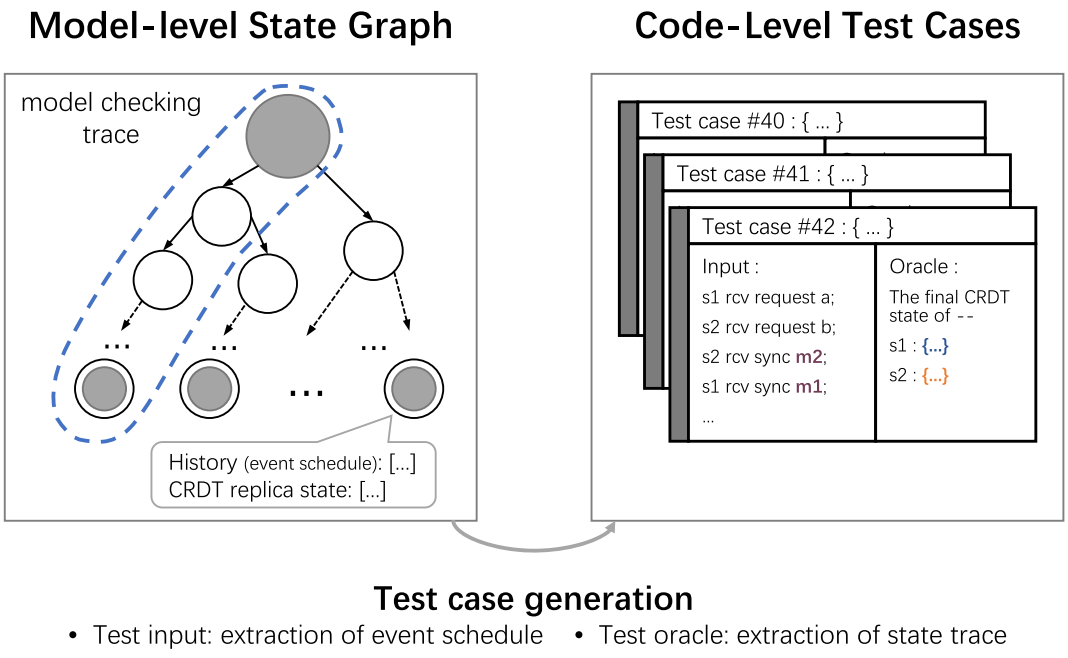

FIGURE 7 From model-level traces to code-level test cases.

4.1.2 | Generation of the test oracle

In MET, the executable code and the TLA+ specification of the CRDT protocol are viewed as just two steps on a ladder of abstraction. Given that the protocol design has been verified by model checking, our objective is to verify that the code-level execution is correct with respect to the model-level execution.

Thus, the sequence of system states in every model-checking trace is extracted to serve as the test oracle.

In explorative testing, we pick one test case and control the system execution to follow the event schedule in the test input.

We check whether the code-level sequence of system states is equivalent to the model-level one. Note that the concrete states of code-level execution contain much more details than the abstract states of model-level model checking. However, the two layers of states contain the same core values. For example, the clock, the existence/value of the data, and the ID of key messages. They are the hooks for us to map the states between the abstract and the concrete layers. Hence, the criteria of being equivalent is that, the metadata for conflict resolution and the concrete data of the data type are the same, while certain details in the code-level states can be ignored.

4.2 | Testability enhancement

The critical challenge in explorative testing is to control the system execution to deterministically follow the event schedule in the test input. Moreover, the test manager also needs to inspect the internal state of the server replica, in order to tell whether the system passes certain test cases. The rationale of testability enhancement in MET is shown in Figure 8. We discuss the two primary issues in testability enhancement in detail below.

4.2.1 | Controllability—Manipulating code-level execution

In TLA+ we model a distributed system in terms of one single global state. Thus, the event schedule in the model-checking trace is a total order of events.

The test manager needs to deterministically control the system execution to follow this total order of events.

The key to achieving system controllability is to control the timing of each event. In the MET framework, we hack the networking component of each server replica. The hacking is transparent to the upper layer protocol. The replicas are provided with the illusion that they communicate with each other via sending and receiving messages. In fact, all the sending and receiving of messages are redirected to the central test manager. The test manager is in full charge of deciding the timing of every event. The system execution makes one step forward when the test manager "fires" one event.

Given the system architecture explained above, we directly integrate the client request generation module into the test manager and do not explicitly model the clients. The test manager will dispatch client request events to the replicas as specified in the test input.

In our application of MET, we instantiate the general approach above over the CRDT-Redis data store.[11]

Details of our implementation can be found in Section 5.1.

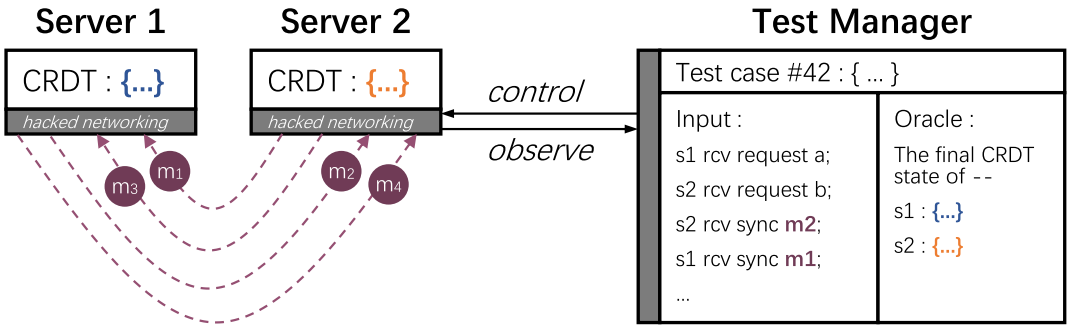

FIGURE 8 System controllability and observability.

4.2.2 | Observability—Inspecting internal state of the replica

Though we can obtain part of the replica state via its public API, the MET framework requires much more than that. As indicated by the test oracle, the test manager requires detailed metadata for conflict resolution. It also requires the internal representation of the data structure.

The commonly used approach is to obtain the required data through extensive logging. However, this logging will require a detailed description of the timing information. It then requires a nontrivial offline analysis of the log.

The MET framework requires the data type store to be enhanced with replica state inspection APIs. Such APIs are dedicated to the test manager whenever it needs to inspect and record the internal state of the server replica.

Given that any progress of the system execution is under the control of the test manager, the test manager can use the APIs for replica state inspection and obtain the system state after every event is dispatched to the replicas. Thus, the state sequence of system execution directly corresponds to the test oracle.

This approach greatly eases the conformance checking against the test oracle.

5 | APPLICATION OF MET IN PRACTICAL CRDT DEVELOPMENT

In this section, we present how we apply the MET framework to cope with deep bugs in our CRDT design and implementation.

We first present the implementation of MET. Then, we discuss how the deep bugs are found and fixed in both the model layer and the code layer.

In the end, we discuss how MET improves our confidence in the correctness of our CRDT design and implementation.

5.1 | Implementation of the MET framework

We specify our CRDT protocols in PlusCal,[29] which is automatically translated to TLA+ specifications. The Rwf-RPQ protocol has about 200 lines of PlusCal code, and Rwf-List about 300.

We model check and revise the design until no violations can be detected. Every model-checking trace of the final design is parsed to generate the test case. The test case generation tool has about 100 lines of Python code.

Our implementation of the Rwf-RPQ and Rwf-List are over the CRDT-Redis data type store we developed.[11] To enhance the testability of CRDT-Redis, we first hack the networking module of the server replica.

Then, we augment each data type with APIs dedicated to the test manager for replica state inspection. The testability enhancement is implemented in about 200 lines of C code.

The test manager is implemented in about 1000 lines of C++ code. The test manager is in charge of launching the server replica groups.

It will intercept all communications among the servers and strictly control the system execution according to the test input.

During the (controlled) system execution, the test manager inspects the internal states of the replicas at its will.

In our practice, we just inspect the final states of the replicas when all messages are delivered to reduce the testing cost.

This simplification is reasonable because in the system state, we record all the metadata for conflict resolution and all data of the data type itself. Our experiences in CRDT development convince us that any divergence in the previous states will be reflected in the final state, and the divergence will be amplified by more data updates. The probability that the divergence is counteracted by following updates is negligible.

The experiment is conducted on a PC with an Intel I7-6700 quad-core CPU (3.40 GHz) and 16 GB RAM. We use VMware Workstation 16 Pro to run the virtual machines. Each VM is set to have a quad-core CPU with 8 GB RAM, running Manjaro with Linux kernel 5.10. The model checking is conducted with TLA+ Toolbox v1.7.1.[30]

All implementations discussed above are available in our GitHub repository online.[11]

We use TLA+ to model the CRDTs and then run TLC model checker to check the design. From the state traces of the output of model checking, we generate the test case to test the CRDT-Redis system. The system is executed under control.

With the help from MET, we find and fix seven bugs in our CRDT design and implementation. Table 1 summarizes the bugs. Note that we mainly focus on relatively deep bugs here. Bugs which can be detected by standard testing techniques are omitted. The bugs found in our application of MET are listed as Issues #1–#7 in the issue page of the online repository.[11] As discussed in Section 2.1.1, Issue #1 can be found by stress testing, but with MET, we find and fix it at a much lower cost. When we use MET to fix Issue #1, we further raise Issue #3, which requires the revision of our CRDT design.

Issue #2 can be detected by manual explorative testing. MET principally automate this process, at a much lower cost.

For Issues #4–#7, they can only be detected by MET.

We discuss these bugs in detail below.

Moreover, MET test the system exhaustively, giving us confidence about the correctness of the CRDT design and implementation.

TABLE 1 The deep bugs we found using MET.

ID	Type	Symptom
Issue #1[15]	Design bug	Inconsistent replicas
Issue #2[18]	Implementation—Assumption bug	Service block
Issue #3[31]	Design bug	Ambiguous list semantics
Issue #4[16]	Implementation—Transcription bug	Unexpected RPQ and list behavior
Issue #5[32]	Implementation—Transcription bug	Unexpected list behavior
Issue #6[33]	Implementation—Transcription bug	Read null element from list
Issue #7[34]	Implementation—Assumption bug	Wrong element order in list

5.2 | Model checking of our CRDT design

We detect Issue #1 by hours of stress testing. We then take days to fix the issue.

Using the MET framework, we first precisely specify the CRDT protocol in TLA+.

The formal specification and model checking help us find bugs deep in subtle corner cases in our design.

The fixing of Issue #1 using MET is much more efficient. The model checking provides the trace which leads to the violation of the correctness condition. This is the shortest trace because the TLC model checker traverses the state space in the BFS manner. We use this trace to help revise the design. We keep revising the design until the design passes the model checking.

The bugs found by the model checking have the following characteristics.

Such bugs can only be triggered by specific interleavings of client requests and replica synchronization operations. The bug-triggering event pattern may not be quite long, but it hides in an exponential number of all possible such patterns. More importantly, the bugs are often latent, in the sense that they may not cause externally observable symptoms for quite a long period of time even after the divergence has appeared.

Thus finding such bugs often requires long time random stress testing. All these characteristics of the bugs necessitate the formal specification and verification of our CRDT design.

The formal specification process in the fixing of Issue #1 drives us to further improve our design, as shown in Issue #3.

In our original design, we treat the *re-add* operation as a special case in the *add* operation. The semantics of the *add* operation becomes quite complex.

Issue #3 basically says that the design should be revised. The *re-add* logic should be treated as a separate operation. The conflict handling logic involves the new operation *re-add* should be supplemented.

Though we put more efforts in designing the new operation *re-add* and the conflict resolution logic involving *re-add*, the overall complexity of our CRDT design is reduced and the resulting specification is less complex. Separating the *re-add* operation out also helps reduce the model-checking cost.

Issue #3. The *re-add* operation should be separated from the *add* operation and treated as an independent operation.[31]

Summarizing the process of fixing Issues #1 and #3, the formal specification forced us to eliminate the ambiguity in semantics of element existence corresponding to the *re-add* operation.

We explicitly model the existence of one element into three cases: *existent*, *non-existent*, and *once-existent*. The meanings of the "existent" and "non-existent" states are as usual. The state "once-existent" pertains to the *re-add* operation. When we *re-add* an element *a*, this element should not be treated as a new element. The *re-add* operation adds an element which once existed in the data container but is now not in the container. We model this status as "once-existent" to correctly design the semantics of the *re-add* operation.

5.3 | MET of our CRDT implementation

Having verified the CRDT protocol in the model layer, we further conduct MET in the code layer. We aim at finding bugs due to transcription errors, named *transcription bugs*, and bugs due to assumptions made in the protocol, named *assumption bugs*.

5.3.1 | Transcription bugs

We use Issue #4 as an example of transcription bugs in Section 2. This bug is beyond the coverage of random stress testing and can only be found by MET.

When transforming the design, both formal and informal, into codes, the developers inevitably needs to handle details which are not covered in the design.

Issue #4 is a bug caused by the deviation between our design and implementation, corresponding to the "dummy-existence" semantics of data elements.

Specifically, at the design level, the element is either exiting or not-existing. The handling is protected by the precondition that "the element must exist." This implicitly states that, if the element does not exist, nothing should be done.

However, at the code level, we must model the intermediate state that the element exists as a dummy. This intermediate state is mainly caused by the late arrival of an operation adding some element into the data container.

In our buggy implementation, the dummy state is not explicitly modeled and illegal positions are assigned to the dummy elements (see details of the example in Section 2.1.2).

Issues #5[32] and #6[33] are similar to Issue #4 in the sense that they are also due to the misinterpretation of the design, concerning the existence of elements. We omit the detailed discussions on these two issues due to the limit of space.

5.3.2 | Assumption bugs

The assumption bug Issue #2 is introduced when we integrate different types of CRDT protocols in the CRDT-Redis platform. Different protocols have different assumptions about the underlying network. There is often a gap between what the protocol assumes and what the platform provides. This significantly increases the odds of assumption bugs in our CRDT implementations. Moreover, we find that implementing a patch fixing such bugs is also error prone and needs extensive testing (see details of this issue in Section 2.1.2).

Issue #7 is another example of an assumption bug. The RWF-List and the RemoveWin-List protocols assume that there is a position ID generator. It generates list element IDs which are totally ordered, dense, and consistent with the element order resulting from the *add* operations. Our design of list element organization (borrowed from the existing design in Weiss et al.[35]) does not specify how such an ID generator can be implemented. The implementation of such an ID generator is nontrivial, and the bug in our implementation of the ID generator is found by MET.

Given the test case execution trace, as well as how the trace violates the test oracle, we quickly find the root cause and revise our implementation of the ID generator.

Issue #7. The order of elements in the RemoveWin List and the RWF-List is sometimes not consistent with the order induced by *add* operations.[34]

5.4 | Evaluation on the effectiveness of MET in practice

Our primary goal of introducing the MET framework is to obtain correct CRDT designs and implementations. In the ideal case, the design has been verified to be correct by model checking. As for the implementation, the testing is both explorative and exhaustive, driven by model checking. Thus the implementation is also correct.

However, in practice, the design and implementation obtained using MET still potentially have bugs. There are mainly two sources of threats to the correctness of CRDT designs and implementations ensured by MET. One is that any details in the implementation which are not modeled in the formal design can still cause bugs. The other is that the state space is not really traversed. A significant portion of the state space is pruned due to the state space explosion problem. This pruning can potentially miss bugs in the design and implementation.

We discuss in detail below how MET provides us with sufficient correctness of CRDT designs and implementations, in spite of the threats discussed above.

5.4.1 | Threats from system modeling

There are always certain aspects of a system operating in realistic scenarios which are not modeled in (both informal and formal) system design. Thus bugs corresponding to the unmodeled part of the system cannot be detected by MET. The unmodeled part of the system is principally the "unknown unknowns."[36–38] We argue that such bugs will not hamper the usefulness of MET. MET is mainly targeted at the subtle deep bugs in CRDT design and implementations. The deep bugs we focus on are primarily "known unknowns"[36–38] as detailed below.

According to our experiences in CRDT design and implementation, deep bugs are mainly known unknowns in the sense that we know or highly suspect their existence. We also know the high-level pattern and mechanism of how the deep bugs may manifest themselves. In theory, the developer can find and fix all the deep bugs through manual reasoning and testing. This approach just requires too much human efforts and time. We lack the tool support to automate and accelerate this manual process, effectively reducing the testing cost.

Our MET framework fills this gap. The gist of proposing the MET framework is thus to automate the most tedious and repetitive part of human reasoning and manual explorative testing. Given the automation provided by MET, human knowledge of the deep bugs can be efficiently leveraged, to dig out the deep bugs at a reasonable cost.

In summary, though MET is no better than the human knowledge of deep bugs, it efficiently leverages knowledge of the known unknowns and pragmatically finds and fixes deep bugs. In this regard, MET can provide us with sufficient confidence in the correctness of CRDT designs and implementations.

5.4.2 | Threats from state space pruning

The total cost for MET is mainly decided by the cost for the model checking. It is because the cost for the explorative testing is proportional to the number of test cases, which is the same as the number of model-checking traces.

We have to limit the scale of the model to finish the testing within the testing budget. The limit on the model scale is based on the small scope hypothesis, as discussed in Section 3.2.

Table 2 shows the different parts of the cost for testing via the MET framework. The configurations shown in the table are the largest affordable scale of models. The MET configuration tuple stands for "number of servers—number of client requests." In our experiment, we limit the testing time by 12 h and limit the disk space used for storing the model-checking traces by 100 GB.

The model checking will not complete if we increase any dimension of the configuration. For example, we tried to increase one client request to the RWF-RPQ model configuration 3-3, which is 3 servers + 4 client requests. The model checking will not finish after 24 hours. Looking at the number of pending states in the queue, we estimate that it needed at least another 24 h. It has used about 150 GB of disk space when we manually stopped the checking.

In the model checking, we have at most three server replicas. The case of one server is used to exercise the basic logic of the protocol without any concurrent updates. The case of two servers is used to test the commutative properties of possible concurrent operations. The case of three servers is used to test the subtle corner cases in conflict resolution. As proved in conflict resolution with the operational transformation (OT) technique in collaborative editing, three operations are enough to express all the properties required for eventual convergence.[26,27]

Thus, we think that the model checking with three replicas can find most deep bugs pertaining to the conflict resolution.

Also note that we have three types of operations for each CRDT.

Thus, we have at least three client requests for each test case, where all types of CRDT operations can be tested.

As shown in Table 2, the cost for Rwf-RPQ and that for Rwf-List differ quite a lot. This is mainly due to the fact that the Rwf-RPQ is data-independent, while the Rwf-List is not. Our checking for the Rwf-RPQ focuses on one data element. We choose $\{10,20\}$ to be the set of possible initial values of *add* operations, and $\{-3,4\}$ the set of possible value increases for *increase* operations. All values are picked uniformly at random.

The Rwf-List does not have the property of data independence. The order of the elements is important for the list. Because each client request may add a new element to the list, we set the size of the element space to the total number of client requests. As we target at collaborative editing scenarios, each data element can have up to six attributes (e.g., the font, being italic, and being bold face). In our testing via MET, we consider only one attribute because we mainly focus on the conflict resolution logic. We choose $\{10,20\}$ to be the set of possible initial values and update values for the Rwf-List.

We also propose an optimization concerning the model-checking trace. We did not let the TLC model checker output the whole state transition graph like the work in Davis et al.[39] Instead, we let the CRDT models output only the final states. This makes the output file one or two orders of magnitude smaller.

The extracted test case files are about half the size of the TLC output files.

The optimization prevents the space usage from being the bottleneck.

TABLE 2 The cost for explorative testing using MET.

			Space cost		Time cost	
	MET config	Num of traces	TLC output	Test case	checking	Testing
Rwf-RPQ	1-10	9,765,625	2.87 GB	1.22 GB	6:30 s	3:10:46 s
	2-5	7,202,312	1.80 GB	884 MB	9:26 s	2:29:03 s
	3-3	883,413	209 MB	105 MB	1:53 s	20:27 s
Rwf-List	1-5	69,343,957	9.64 GB	5.47 GB	2:43:22 s	11:50:55 s
	2-4	23,639,180	4.09 GB	2.1 GB	2:12:38 s	5:09:29 s
	3-3	5,085,147	1.43 GB	731 MB	51:05 s	1:46:18 s

To get a rough idea, it takes about one hour to test 1 GB of test case data on average.

Comparing the time cost for the model checking and that for the explorative testing, we find that the model checking is faster than the explorative testing. This is mainly because the model checker can prune the redundant checking of system states, while in MET the execution of each test case is stateless.

According to the discussions above, the cost for testing via MET is reasonable. Now, the critical issue is to discuss whether MET can help us effectively find deep bugs in our CRDT design and implementation. The effectiveness of the MET framework depends on the small scope hypothesis. We argue that the hypothesis works well in the testing of CRDT implementations. In collaborative editing scenarios, convergence properties involving at most three concurrent operations are sufficient to ensure eventual convergence. Because OT techniques also focus on ensuring eventual convergence among replicas, this result indicates that with three replicas we can construct the triggering pattern of most deep bugs in CRDT development. This result is in accordance with our experiences in CRDT testing. As discussed in our design of MET, we focus on the conflict resolution logic of CRDTs. We leverage our understanding of the CRDT design to carefully prune parts of the state space which are not relevant or less relevant to the conflict resolution logic.

In summary, most deep bugs concerning the conflict resolution logic can be reproduced in small scale systems. The pruning in our model checking in MET carefully avoids erroneously pruning traces which can trigger deep bugs. In this regard, we believe that MET ensures sufficient correctness of CRDT designs and implementations.

6 | RELATED WORK

6.1 | Model-checking-distributed protocols

Distributed systems are notoriously difficult to implement correctly, mainly due to the intrinsic uncertainty in their executions.[40] This uncertainty is created by asynchrony, the absence of global time, various types of failures, and so on. From the perspective of adversary argument,[41] the developer needs to enumerate all possible combinations of uncertain events and reason out the invariable correctness of system behavior. This type of exhaustive reasoning is often impractical for the developer. This explains why model-checking techniques are often employed to verify the correctness of distributed system designs.[14,42]

For example, the Chord protocol is formally specified in Alloy and is model checked using the Alloy Analyzer.[43,44]

Distributed consensus protocols, such as Paxos[45] and Raft,[46] are specified in TLA+ and model checked using the TLC model checker. TLA+ is also widely used to verify the design of distributed systems in industrial scenarios, for example, in Amazon S3 and DynamoDB,[14,47] CosmosDB,[48] and TaurusDB.[49] The techniques discussed above mainly focus on design at the model level.

The MET framework leverages model checking to further improve the reliability of code-level implementations.

6.2 | Distributed system model checking (DMCK)

DMCK treats the code-level implementation as the model and directly conducts model checking on the code.[9,50–52] DMCK precisely controls the execution of the distributed system and exhaustively explores all possible execution paths. DMCK has shown the potential for detecting deep bugs which are extremely difficult to detect through traditional testing techniques. However, DMCK usually imposes a prohibitive cost, and state reduction techniques are essential to the practical application of DMCK.

Partial order reduction exploits the independence among events and establishes the equivalence among interleavings of independent events.[51] Thus, the state space is reduced by checking only one interleaving on behalf of all equivalent ones. Dynamic interface reduction[52] is based on the observation that different interleavings of local events (within one system component) often result in the same global interaction (among multiple components). Thus, the redundant enumeration of global events can be saved.

To better reduce the state space, semantic knowledge of the target system can further be utilized. The semantic-aware model checking (SAMC)[9] presents four semantic-aware reduction policies that enable the model checker to define fine-grained event dependency/independency and symmetry beyond what black-box approaches can do.

Our MET framework shares the same goal of DMCK techniques. We all aim to inherit the feature of exhaustive exploration of model checking to improve code-level testing.

However, the MET framework takes a methodologically different approach. MET does not aim at fully automatic checking of the code-level implementation as the model. MET is designed to be semi-automatic. Human efforts are required to obtain the formal specification of the system design. The developer is also required to enhance the testability of the system under test. After the manual preparations above, the model checking, the test case generation, and the explorative testing are automatic.

We opt for this semi-automatic approach mainly based on the observation that, fully automatic DMCK approaches still imposes a prohibitive cost on distributed systems in the field. We leverage moderate human efforts to reduce the overall testing cost, while still inheriting the exhaustive exploration of model checking to code-level testing. Another advantage of our MET is that we can thoroughly check the correspondence between the design and the implementation. We argue that this tradeoff is effective and efficient for subtle deep bugs in CRDT implementations and hopefully in more distributed systems.

6.3 | Fault injection testing

Fault injection testing focuses on testing the aspect of fault-tolerance of distributed systems.[53–56] For example, the Failure Testing Service (FATE) focuses on recovery testing.[53] FATE exhaustively exercises as many combinations of failures as possible. Semantic knowledge of the failure patterns is utilized to prioritize the failure patterns. The limited testing budget is then directed to the testing of the high-priority patterns. Our MET framework currently focuses on the correctness of the design and implementation of distributed protocols. Theoretically MET can also model the fault-tolerance design and implementation of distributed systems. We intend to do so in our future work.

6.4 | Randomized concurrency testing

Instead of systematically testing the distributed system, randomized concurrency testing tackles the intractable test space via sampling. Probabilistic concurrency testing (PCT)[57,58] and partial order sampling (POS)[59] are two sampling strategies used to test distributed systems these years. They provide probabilistic guarantees to find concurrency bugs with certain patterns at a reasonable cost. MET aims at systematically testing not only the distributed system but also the system design, together with the correspondence between the system and the design. The state explosion problem is alleviated by importing the designer's knowledge to abstract away the irrelevant details. MET further provides certain correctness guarantee for the distributed systems that have passed the testing.

6.5 | Model-based test case generation (MBTCG)

MongoDB uses MBTCG to ensure the equivalence between the C++ and the Golang versions of the OT implementations in MongoDB Realm Sync.[39] MET is inspired by the MBTCG technique of MongoDB. Moreover, the OT technique also focuses on ensuring eventual convergences among different replicas, which motivates us to first apply MET on CRDT designs and implementations.

The MBTCG technique is used at the unit testing level, and focuses on the OT module in the Realm Sync system. The MET framework is targeted at subtle deep bugs in CRDT data stores, and the explorative testing is conducted at the system level.

7 | CONCLUSION

In this work, we present the MET framework to cope with deep bugs in CRDT designs and implementations. The developer first specifies the CRDT design in TLA+, and model checks the design to eliminate model-level bugs. Then, test cases are automatically generated from the model-checking traces to achieve explorative and exhaustive testing at the code level. We demonstrate a practical application of the MET framework in the design and implementation of various data types over the CRDT-Redis data store. The MET framework not only eases the fixing of bugs which can be detected by standard testing techniques. It also finds bugs which are out of reach of existing techniques. We also discuss how MET increases our confidence in the correctness of CRDT designs and implementations.

In our future work, we will apply MET to more different types of distributed systems, for example, distributed consensus components and atomic commit components in cloud-native distributed databases. We will also explore heuristic exploration strategies and parallel checking frameworks, to further tame the state explosion problem in MET.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (62025202) and the Cooperation Fund of Huawei-Nanjing University Next Generation Programming Innovation Lab (YBN2019105178SW38).

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in CRDT-Redis at https://github.com/elem-azar-unis/CRDT-Redis.[11]

ORCID

Yuqi Zhang https://orcid.org/0000-0003-1416-9379

ENDNOTES

* The term eventual consistency may have slightly different meanings in different contexts. In this work, by eventual consistency we mean strong eventual consistency defined in Shapiro et al.[2]

† When considering the conflict resolution, the elements in the Rwf-RPQ can be considered independent. When the server replica locally maintains the priority queue data structure, the data elements are dependent. But this dependence is not relevant to the conflict resolution we focus on in this work.

REFERENCES

1. Burckhardt S, Gotsman A, Yang H, Zawirski M. Replicated data types: Specification, verification, optimality. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '14. ACM; 2014:271-284.
2. Shapiro M, Preguiça N, Baquero C, Zawirski M. Conflict-free replicated data types. In: Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems. SSS'11. Springer-Verlag; 2011:386-400. http://dl.acm.org/citation.cfm?id=2050613.2050642
3. Enes V, Almeida PS, Baquero C, Leitao J. Efficient synchronization of state-based crdts. In: 2019 IEEE 35th International Conference on Data Engineering (ICDE). IEEE; 2019:148-159.
4. Brewer E. Cap twelve years later: how the "rules" have changed. *Computer*. 2012;45(2):23-29.
5. Gilbert S, Lynch NA. Perspectives on the CAP Theorem. *Computer*. 2012;45(2):30-36.
6. Riak. Riak enterprise NoSQL database, scalable database solutions; 2022. https://riak.com/
7. Redis. Redis enterprise, database built for hybrid applications; 2022. https://redis.com/redis-enterprise/
8. Microsoft Azure. Azure Cosmos DB—NoSQL database, Microsoft Azure; 2022. https://azure.microsoft.com/en-us/services/cosmos-db/
9. Leesatapornwongsa T, Hao M, Joshi P, Lukman JF, Gunawi HS. SAMC: semantic-aware model checking for fast discovery of deep bugs in cloud systems. In: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation. OSDI'14. USENIX Association; 2014:399-414. http://dl.acm.org/citation.cfm?id=2685048.2685080
10. The TLA+ home page; 2022. https://lamport.azurewebsites.net/tla/tla.html
11. GitHub. Conflict-free replicated data types based on Redis; 2022. https://github.com/elem-azar-unis/CRDT-Redis
12. Zhang Y, Wei H, Huang Y. Remove-win: a design framework for conflict-free replicated data types. In: Proceedings of the 27th IEEE International Conference on Parallel and Distributed Systems. ICPADS'21. IEEE; 2021.
13. Brown MA. Traffic Control HOWTO; 2020. Accessed September 30 2020. http://tldp.org/HOWTO/Traffic-Control-HOWTO/index.html
14. Newcombe C, Rath T, Zhang F, Munteanu B, Brooker M, Deardeuff M. How amazon web services uses formal methods. *Commun ACM*. 2015;58(4):66-73. https://doi.org/10.1145/2699417
15. GitHub. CRDT-Redis: Issue #1; 2022. https://github.com/elem-azarunis/CRDT-Redis/issues/1
16. GitHub. CRDT-Redis: Issue #4; 2022. https://github.com/elem-azar-unis/CRDT-Redis/issues/4
17. Xu T, Jin X, Huang P, Zhou Y, Lu S, Jin L, Pasupathy S. Early detection of configuration errors to reduce failure damage. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation. OSDI'16. USENIX Association; 2016:619-634.
18. GitHub. CRDT-Redis: Issue #2; 2022. https://github.com/elem-azar-unis/CRDT-Redis/issues/2
19. Shapiro M, Preguiça N, Baquero C, Zawirski M. A comprehensive study of convergent and commutative replicated data types. RR-7506, Inria—Centre Paris-Rocquencourt; INRIA; 2011. Research Report. https://hal.inria.fr/inria-00555588
20. Jackson D. *Software Sbstractions: Logic, Language, and Analysis*: The MIT Press; 2012.
21. Yuan D, Luo Y, Zhuang X, et al. Simple testing can prevent most critical failures: an analysis of production failures in distributed data-intensive systems. In: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation. OSDI'14. USENIX Association; 2014:249-265.
22. Marić O, Sprenger C, Basin D. Cutoff bounds for consensus algorithms. In: Majumdar R, Kunčak V, eds. *Computer Aided Verification*: Springer International Publishing; 2017:217-237.
23. Hance T, Heule M, Martins R, Parno B. Finding invariants of distributed systems: it's a small (enough) world after all. In: 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI '21). USENIX Association; 2021:115-131. https://www.usenix.org/conference/nsdi21/presentation/hance
24. Andoni A, Daniliuc D, Khurshid S, Marinov D. Evaluating the "small scope hypothesis". Citeseer; 2002.
25. Oetsch J, Prischink M, Pührer J, Schwengerer M, Tompits H. On the small-scope hypothesis for testing answer-set programs. In: Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning. KR'12. AAAI Press; 2012:43-53.
26. Sun C, Xu Y, Agustina A. Exhaustive search of puzzles in operational transformation. In: Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work and Social Computing. CSCW '14. Association for Computing Machinery; 2014:519-529.
27. Xu Y, Sun C, Li M. Achieving convergence in operational transformation: conditions, mechanisms and systems. In: Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work and Social Computing. CSCW '14. Association for Computing Machinery; 2014:505-518.
28. Lamport L, Merz S. Auxiliary variables in TLA+. CoRR abs/1703.05121; 2017. http://arxiv.org/abs/1703.05121
29. Lamport L. The PlusCal algorithm language. In: Leucker M, Morgan C, eds. *Theoretical Aspects of Computing—ICTAC 2009*: Springer Berlin Heidelberg; 2009:36-60.
30. GitHub. TLA+—the brontinus release; 2022. https://github.com/tlaplus/tlaplus/releases/tag/v1.7.1
31. GitHub. CRDT-Redis: Issue #3; 2022. https://github.com/elem-azar-unis/CRDT-Redis/issues/3
32. GitHub. CRDT-Redis: Issue #5; 2022. https://github.com/elem-azar-unis/CRDT-Redis/issues/5
33. GitHub. CRDT-Redis: Issue #6; 2022. https://github.com/elem-azar-unis/CRDT-Redis/issues/6
34. GitHub. CRDT-Redis: Issue #7; 2022. https://github.com/elem-azar-unis/CRDT-Redis/issues/7

35. Weiss S, Urso P, Molli P. Logoot-Undo: Distributed collaborative editing system on P2P networks. *IEEE Trans Parallel Distribut Syst*. 2010;21(8):1162-1174.
36. Garlan D. Software engineering in an uncertain world. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research. FoSER '10. Association for Computing Machinery; 2010:125-128.
37. Esfahani N, Kouroshfar E, Malek S. Taming uncertainty in self-adaptive software. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. ESEC/FSE '11. Association for Computing Machinery; 2011:234-244.
38. Elbaum S, Rosenblum DS. Known unknowns: testing in the presence of uncertainty. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. FSE 2014. Association for Computing Machinery; 2014:833-836.
39. Davis AJJ, Hirschhorn M, Schvimer J. Extreme modelling in practice. *Proc VLDB Endow*. 2020;13(9):1346-1358. https://doi.org/10.14778/3397230.3397233
40. Fonseca P, Zhang K, Wang X, Krishnamurthy A. An empirical study on the correctness of formally verified distributed systems. In: Proceedings of the Twelfth European Conference on Computer Systems. EuroSys '17. Association for Computing Machinery; 2017:328-343. https://doi.org/10.1145/3064176.3064183
41. Erickson J. More algorithms lecture notes; 2022. http://jeffe.cs.illinois.edu/teaching/algorithms/
42. Newcombe C. Debugging designs using exhaustively testable pseudo-code. In: Proceedings of the International Workshop on High Performance Transaction Systems. HPTS; 2011. http://hpts.ws/papers/2011/agenda.html
43. Zave P. How to make chord correct (using a stable base). CoRR abs/1502.06461; 2015. http://arxiv.org/abs/1502.06461
44. Zave P. Reasoning about identifier spaces: how to make chord correct. *IEEE Trans Softw Eng*. 2017;43(12):1144-1156.
45. GitHub. Dr. TLA+ Series—Paxos (Andrew Helwer); 2022. https://github.com/tlaplus/DrTLAPlus/tree/master/Paxos
46. GitHub. Formal TLA+ specification of the raft consensus algorithm; 2022. https://github.com/ongardie/raft.tla
47. Newcombe C. Why Amazon chose TLA+. In: Ait Ameur Y, Schewe K-D, eds. *Abstract State Machines, Alloy, B, TLA, VDM, and Z*: Springer Berlin Heidelberg; 2014:25-39.
48. GitHub. High-level TLA+ specifications of the five consistency levels offered by Azure Cosmos DB; 2022. https://github.com/Azure/azure-cosmos-tla
49. Gao S, Zhan B, Liu D, et al. Formal verification of consensus in the Taurus distributed database. In: Huisman M, Păsăreanu C, Zhan N, eds. *Formal Methods*: Springer International Publishing; 2021:741-751.
50. Musuvathi M, Park DYW, Chou A, Engler DR, Dill DL. CMC: a pragmatic approach to model checking real code. In: Proceedings of the 5th Symposium on Operating Systems Design and Implementation. OSDI '02. USENIX Association; 2002:75-88.
51. Yang J, Chen T, Wu M, et al. MODIST: transparent model checking of unmodified distributed systems. In: NSDI'09. USENIX; 2009:213-228.
52. Guo H, Wu M, Zhou L, Hu G, Yang J, Zhang L. Practical software model checking via dynamic interface reduction. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. SOSP '11. ACM; 2011:265-278.
53. Gunawi HS, Do T, Joshi P, et al. FATE and DESTINI: a framework for cloud recovery testing. In: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation. NSDI'11. USENIX Association; 2011:238-252.
54. Alvaro P, Rosen J, Hellerstein JM. Lineage-driven fault injection. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. SIGMOD '15. Association for Computing Machinery; 2015:331-346.
55. Lu J, Liu C, Li L, et al. CrashTuner: detecting crash-recovery bugs in cloud systems via meta-info analysis. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles. SOSP '19. Association for Computing Machinery; 2019:114-130.
56. Chen H, Dou W, Wang D, Qin F. CoFI: consistency-guided fault injection for cloud systems. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. ASE '20. Association for Computing Machinery; 2020:536-547.
57. Ozkan BK, Majumdar R, Niksic F, Befrouei MT, Weissenbacher G. Randomized testing of distributed systems with probabilistic guarantees. *Proc ACM Program Lang*. 2018;2(OOPSLA):1-28. https://doi.org/10.1145/3276530
58. Ozkan BK, Majumdar R, Oraee S. Trace aware random testing for distributed systems. *Proc ACM Program Lang*. 2019;3(OOPSLA):1-29. https://doi.org/10.1145/3360606
59. Yuan X, Yang J. Effective concurrency testing for distributed systems. In: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. Association for Computing Machinery; 2020:1141-1156.

How to cite this article: Zhang Y, Huang Y, Wei H, Ma X. Model-checking-driven explorative testing of CRDT designs and implementations. *J Softw Evol Proc*. 2024;36(4):e2555. doi:10.1002/smr.2555