ER-π: Exhaustive Interleaving Replay for Testing Replicated Data Library Integration

Provakar Mondal and Eli Tilevich
Software Innovations Lab, Virginia Tech, USA
{provakar,tilevich}@cs.vt.edu

Abstract

Modern replicated data systems often rely on libraries integrated with application code. These replicated data libraries exchange asynchronous messages, whose execution orderings are non-deterministic, allowing any message interleaving to occur during system execution. Testing the integration of application code with library code requires considering all possible interleavings, whose detection and simulation pose significant challenges for application developers. In this paper, we present ER-π, a middleware system, designed to detect and replay possible interleavings in replicated data systems. ER-π identifies potential interleavings for a given code segment and applies four novel pruning techniques to reduce the complexity of the problem space. Subsequently, it replays the remaining interleavings to perform the specified integration testing tasks. To assess the applicability and efficacy of ER-π, we integrated it with third-party replicated data libraries across various programming languages. Our experiments demonstrate ER-π 's capability to replicate 12 known bugs and uncover 5 types of common misconceptions associated with replicated data libraries. Given that integration testing is essential for ensuring correctness and robustness, the design of ER-π holds promise in extending these testing benefits to the realm of replicated data systems.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Keywords: Replicated Data System, Integration Testing, Middleware Service, Exhaustive Replay

ACM Reference Format:
Provakar Mondal and Eli Tilevich. 2025. ER-π: Exhaustive Interleaving Replay for Testing Replicated Data Library Integration. In *Proceedings of 26th ACM/IFIP International Middleware Conference (Middleware 2025).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

As the demand for replicated data systems keeps increasing, the software development infrastructure features expressive and intuitive libraries that provide robust and versatile conflict resolution. These libraries follow some consistency protocol, among which Eventual Consistency has become widely used [58]. Application developers integrate these third-party libraries into their applications to manage replicated data. As a result, developers focus on application business logic, while relying on the libraries for replicated data management. Integration testing is required to test the correctness of interactions between the application and the library.

With their eventual consistency for managing replicated data, powerful third-party libraries may give application developers a false sense of confidence. The developers may incorrectly assume that because the replicated data library (RDL)[1] guarantees eventual consistency, the interactions across the library and the developer-provided application logic will be correct. However, consistency is not synonymous with correctness, so subtle bugs can arise from scenarios that include wrong usage of the RDLs, incorrect assumptions in application logic, incorrect synchronization, or ill-conceived data models on top of the library [24]. Furthermore, a recent study has revealed that application developers may misunderstand certain properties of third-party RDLs [57]. Consequently, interactions between RDLs and application code can contain subtle bugs.

Such subtle bugs can only be detected by exhaustively testing the system across possible interleavings [38]. Furthermore, integration testing is needed to ensure that the application code interacts correctly with the underlying data library. Certain bugs such as destructive data races, consistent with incorrect results can manifest themselves only during specific interleavings [44]. To fix a reported bug, developers should be able to reproduce it [40]. Reproducing bugs in replicated data systems requires tools that provide an exhaustive replay of possible interleavings.

In this paper, we present **E**xhaustive **R**eplay of **P**ossible **I**nterleavings or ER-π for short, a middleware service that enables applications to perform integration testing with all

[1]Hereafter, we refer to the Replicated Data Library as RDL

possible interleavings of interactions with the RDL. Having been applied to the application, ER-π determines the events invoked from the RDL via language-specific proxying. Using the constraints applicable to the particular RDL being used, ER-π applies four novel pruning algorithms to reduce the number of interleaving to replay. ER-π persists the resulting interleavings and executes them without requiring any modification to the RDL. To ensure the current order of events in each interleaving, ER-π uses a distributed locking mechanism. The majority of ER-π's functionality is language agnostic, including the generation of exhaustive interleavings, their pruning, and coordinating their execution using a distributed lock. We have applied ER-π to 5 open-source existing applications that use RDLs to maintain their data. Our approach has successfully reproduced 12 priorly reported bugs and found 5 types of library usage misconceptions, thus demonstrating the effectiveness and usefulness of ER-π. By describing ER-π's design, implementation, and evaluation, this paper makes the following contributions:

1. A novel integration testing approach that verifies if the distributed application logic correctly interacts with an RDL; the approach determines and executes the possible interleavings of distributed events raised by invoking the library functions.
2. Four novel pruning algorithms that reduce the number of possible interleavings by applying the unique execution constraints of replicated data systems.
3. An implementation of our approach as ER-π, a middleware framework that interfaces with applications code to detect and replay the possible interleavings of the distributed events. ER-π features
   a) a language-independent part that works with RDLs implemented in compiled, interpreted, and managed languages.
   b) a library of test functions, through which programmers can execute various tests.
4. An empirical evaluation of ER-π that shows its effective application across several third-party RDLs, its ability to reproduce previously reported bugs, and identify key known misconceptions about integrating the RDLs with application code.

The rest of the paper is organized as follows: Section 2 describes the background and motivates the problem; Section 3 explains the pruning algorithms of ER-π to reduce problem space; Section 4 presents our system design and workflow; Section 5 describes the technologies to build and apply ER-π to the evaluation subjects; Section 6 presents our evaluation results, demonstrating the effectiveness and usefulness of ER-π for integration testing; Section 7 describes the related state of the art; and finally Section 8 concludes the paper with future work plans.

2 Background and Motivation

In this section, we first provide the background of replicated data systems and systematic testing with different interleavings. Then we motivate the need for exhaustive replaying with possible interleavings for eventual consistent replicated data systems.

2.1 Replicated Data System

Data replication creates and maintains multiple copies of the same data (known as replicas) in different locations as a way of ensuring data availability, low latency, and avoiding a single point of failure [37]. Modern users typically possess multiple computing devices, including smartphones, tablets, smartwatches, and desktop computers, through which they often access the same data, which can also be shared with other users. As a result, this data needs to be replicated across multiple devices, each of which might modify the data replicas independently. Network disconnections and faults often make it impossible to synchronize all the updates immediately. As a result, eventual consistency protocols have become a pragmatic and effective approach for managing replicated states in modern distributed applications. Amazon's DynamoDB [19], Microsoft's Azure Cosmos DB [46], Apache Hadoop [51], etc. are examples of platforms that provide data replication support. Among the family of algorithms for optimistic data replication, Conflict-Free Replicated Data Types (CRDTs) [45], Explicitly Consistent Replicated Objects (ECROs) [17], Mergeable Replicated Data Types (MRDTs) [27] are some popular solutions.

2.2 Testing Interleavings in Distributed Systems

Our work is an example of a distributed model checker (DMCK), a tool that interleaves the non-deterministic events to push the target system into unexplored states, thus revealing hard-to-find bugs [50]. Interleavings represent the different possible sequences in which events can occur and how they interact with one another. Testing different interleavings helps uncover potential race conditions, deadlocks, or inconsistencies that may arise due to the concurrent execution of events within the distributed environment [34].

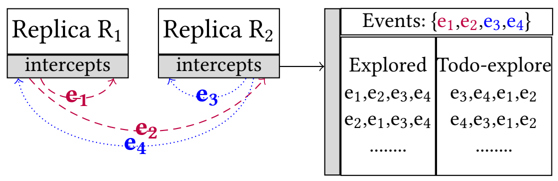


Figure 1. Typical DMCK's Work Strategy

Figure 1 shows how a basic model checker can work with a replicated data system comprising the replicas, R_1 and R_2. By interception, the checker detects the events executed in a particular workload. In this example, the events executed during the intercepting time are e_1, e_2, e_3, and e_4. In the

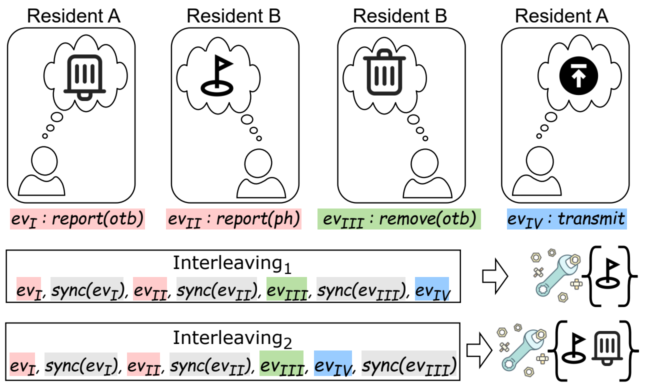

Resident A
Resident B
Resident B
Resident A
ev_I : report(otb)
ev_{II} : report(ph)
ev_{III} : remove(otb)
ev_{IV} : transmit
Interleaving$_1$
ev_I, sync(ev_I), ev_{II}, sync(ev_{II}), ev_{III}, sync(ev_{III}), ev_{IV}
Interleaving$_2$
ev_I, sync(ev_I), ev_{II}, sync(ev_{II}), ev_{III}, ev_{IV}, sync(ev_{III})

Figure 2. Motivating Example: Events and Interleavings

absence of any path reduction algorithm, a model checker can use Depth-First Search (DFS) for path exploration, and it will return 4! interleavings to explore for four events. The checker's server will keep track of which interleavings have been explored and which are still on the to-do list to explore. Before running a new interleaving, the checker will restart the workload, so that no interleaving can affect others. The developers can decide which global properties to check (e.g., replicas' states, consistency, memory usage, etc.). After executing each interleaving, the checker runs the specified test functions to determine any property violations.

2.3 Motivating Example

Dr. Strange:	I went forward in time to view alternate futures. To see all the possible outcomes of the coming conflict.
Star Lord:	How many did you see?
Dr. Strange:	Fourteen million, six hundred and five.
Iron Man:	How many did we win?
Dr. Strange:	[long beat] One!

Movie: [Avengers: Infinity War] [39]

As mentioned in Section 1, eventual consistency does not eliminate the need for integration testing, which must consider all possible interleavings. Consider a town's administration providing a free mobile app for residents to report issues that need fixing. The app allows users to report a problem (with geolocation) and remove a reported problem once it has been fixed. Reported issues are represented as a replicated set to avoid duplicate entries.

Imagine two residents using the app simultaneously: `Resident A` reports an overturned trash bin, and `Resident B` reports a pothole. Each app user holds a replica, and all problems, added or removed, are synchronized across replicas. Then, `Resident B` observes that a custodian has fixed the trash bin reported by `Resident A` and removes the overturned trash bin issue from the set of problems. When residents stop using the app, they can transmit their set of problems to the municipality for fixing. Let's assume `Resident A` transmits this set of problems to the municipality.

This scenario, depicted in Figure 2, involves seven distributed events: (1) ev_I: reporting an overturned trash bin (`otb` in the figure) by Resident A; (2) $sync(ev_I)$: synchronization of ev_I to Resident B; (3) ev_{II}: reporting a pothole (`ph` in the figure) by Resident B; (4) $sync(ev_{II})$: synchronization of ev_{II} to Resident A; (5) ev_{III}: removing the overturned trash bin report of ev_I by Resident B; (6) $sync(ev_{III})$: synchronization of ev_{III} to Resident A; and (7) ev_{IV}: transmitting the set of reported problems to the municipality by Resident A. Suppose the app developers assumed, based on the erroneous assumption—"multiple replicas in different regions mathematically resolve to the same state without coordination" [26]—that eventual consistency makes further conflict resolution unnecessary. However, without coordination, depending on the event order, the municipality may receive a set of problems that either include (1) only the pothole or (2) both issues, while the correct outcome is (1). Specifically, `Resident A` can transmit the list of identified problems before or after synchronizing the update from `Resident B` about the fixed trash bin. Consider two possible interleavings, only the first of which leads to the correct outcome: (1) ev_I, $sync(ev_I)$, ev_{II}, $sync(ev_{II})$, ev_{III}, $sync(ev_{III})$, ev_{IV}; (2) ev_I, $sync(ev_I)$, ev_{II}, $sync(ev_{II})$, ev_{III}, ev_{IV}, $sync(ev_{III})$.

The incorrect outcome arises not because something is amiss with the RDL or the application logic, but rather how these two interact following the erroneous assumption. Detecting such design flaws requires testing the system against all possible event interleavings. By expressing our motivating example as a test case and defining a test invariant that only the pothole issue is transmitted, we can identify the interleavings that violate the invariant (e.g., Interleaving$_2$).

Our middleware system—ER-π—solves this problem by determining possible interleavings of the events. For example, in theory, the seven events of the motivating example can interleave in 7! = 5040 ways. However, for this example, ER-π's pruning algorithms reduce the possible number of interleavings to 19, thus reducing the problem space by $\lfloor \frac{5040}{19} \rfloor = 265$ times. We provide the details of this particular pruning scenario in Section 3.1. As the factorial function is fast growing, the search space increases rapidly with more events (i.e., 10 events result in over 3.6 million interleavings). Hence, ER-π's four pruning algorithms are required to reduce the search space and make exhaustive replay practical. After determining possible interleavings and with the given test invariants, ER-π exhaustively tests the invariants against all interleavings, reporting any violations found.

3 ER-π's Pruning Algorithms

In this section, we describe ER-π's pruning algorithms designed to reduce the number of interleavings to replay. These algorithms consider the constraints specific to the nature of RDL. To understand how pruning works in action, let us revisit the pruning in the motivating example.

3.1 Pruning in Motivating Example

Recall that the example in Section 2.3 mentions pruning the number of interleavings to replay from 5040 to 19. One pruning strategy of ER-π is grouping the `sync` events with the corresponding `update` events, as synchronization causally depends on updates. To interleave the synchronization event before the update event would be unreasonable. Grouping $(ev_I, sync(ev_I))$, $(ev_{II}, sync(ev_{II}))$, and $(ev_{III}, sync(ev_{III}))$ creates three paired events, so with the remaining one, the total number of events become four, which interleave in $4! = 24$ ways. Now, interleaving ev_{IV} into the first position would always cause the empty set of problems to be sent to the municipality. With ev_{IV} in the first position, the following three events can interleave $3! = 6$ ways, each of which causes the empty set. So, instead of considering them individually, they can be merged into a single interleaving, thus pruning $6 - 1 = 5$ more interleavings. As a result, the final number of interleavings becomes $24 - 5 = 19$, which should be quite manageable to replay exhaustively.

We equip ER-π with four pruning methods. For initial pruning, ER-π applies `Event Grouping` and `Replica Specific` pruning. Upon generating and executing the interleavings one by one, application developers can discover some properties of the events by analyzing their effects. Then, parameterized by developers, ER-π applies `Event Independence` and `Failed Ops` pruning to reduce the problem space further. In the following subsection, we present each of the pruning algorithms in the following format: (1) the intuition for the pruning, (2) an example, and (3) the pseudo-code of the implementation.

3.2 Event Grouping

Intuition: ER-π groups together `send sync request` event with `execute sync request` event in the (same sender, receiver) pair as the second event is followed by the first event and the second event will not occur if the first event has not taken place. Interleaving them in different orders would be wasteful: only if a replica has sent a sync request, the receiver replica would be able to execute it then. Furthermore, ER-π can group events if explicitly directed to do so by the user.

Example: Consider the example depicted in Figure 3. Between two replicas, A and B, there are eight events in total. In theory, these events can interleave in $8! = 40320$ ways. However, as one can see from the Figure, ev_3 denotes sending a synchronization request from replica A to replica B, and ev_4 indicates that replica B executes the sync request. Similarly, ev_7 sends a sync request from replica B to A, and in ev_8, replica A executes the corresponding synchronization. With the event-grouping algorithm in action, ER-π will group ev_3 with ev_4 and ev_7 with ev_8, thus reducing the total number of events to six. Pruning based on event grouping alone would reduce the number of interleaving by $\frac{8!}{6!} = 56$ times. Algorithm 1 shows the step-by-step pseudo-code that applies the event-grouping pruning logic.

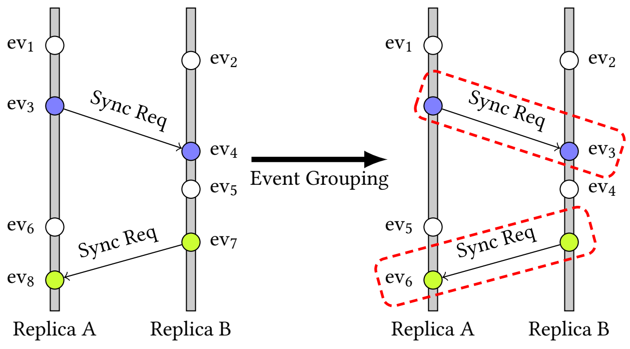


Figure 3. Grouping Events to Reduce Their Total #

Algorithm 1: Event Group Pruning

```
Input: $events$: events to interleave
Input: $spec\_group$: developers' specified groups
Output: $\overline{GI}$: interleavings based on grouping

$events \leftarrow read\_events()$
$spec\_group \leftarrow read\_specified\_groups()$
$grouped\_events = []$                    // list of tuple
$\overline{GI} = []$        // list of group-by pruned interleavings
forall $(event_i, event_j) \in events$ do
    if $(event_i \equiv sync\_req \wedge event_j \equiv exec\_sync) \vee$
      $(event_j \equiv sync\_req \wedge event_i \equiv exec\_sync)$ then
        $from_i \leftarrow event_i.fromReplicaId$
        $to_i \leftarrow event_i.toReplicaId$
        $from_j \leftarrow event_j.fromReplicaId$
        $to_j \leftarrow event_j.toReplicaId$
        if $from_i = from_j$ & $to_i = to_j$ then
            $grouped\_events.append(\{event_i, event_j\})$
forall $(event_m, event_n) \in spec\_group$ do
    $grouped\_events.append(\{event_m, event_n\})$
    /* grouping by developers' specified info */
$\overline{GI} = permute(events, grouped\_events)$
return $\overline{GI}$
```

3.3 Replica Specific

Intuition: Replica-specific exploration refers to ascertaining a behavior specific to a particular replica. To facilitate such exploration, ER-π groups together those events executed at other replicas that have no impact on the state of the explored replica. If developers want to do integration testing for a particular replica, then the events executed at other replicas without impacting the tested replica can be grouped. In such scenarios, the interleavings, which are permutations of the grouped events, can be removed from the total number of interleavings.

Example: Consider exploring the behavior of replica B, as depicted in the example in Figure 4. In replicated data

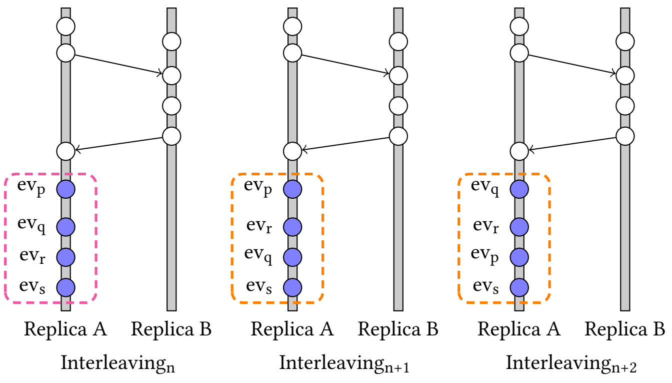


Figure 4. Replica B-specific Pruning

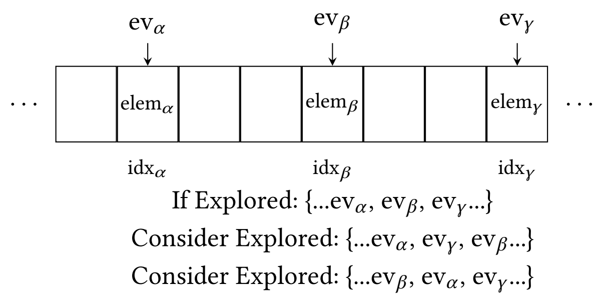
systems, a replica can only see the impact of updates at the other replicas only having synchronized the update requests from these replicas. In this example, the occurrence of the events ev_p, ev_q, ev_r, and ev_s at replica A would not impact replica B if these events are interleaved after the last sync request from replica A to replica B. In such interleavings, considering a different order of these four events would be wasteful. Hence, the interleaving$_n$, interleaving$_{n+1}$, and interleaving$_{n+2}$ can be merged. With this pruning, ER-π can reduce the total number of interleavings by $4! - 1 = 23$. Algorithm 2 shows the step-by-step pseudo-code that applies replica-specific pruning logic.

Algorithm 2: Replica-specific Pruning

```
Input: $ILs$: list of existing interleavings
Input: $rID$: specific replica ID
Output: $\overline{RI}$: interleavings pruned by replica-specific
1  $ILs \leftarrow read\_interleavings()$
2  $grouped\_events = []$                                   // a list
3  $grouped\_by\_indices = \{\}$
   /* a map, (key,value) ≡ (indices,interleaving)        */
4  $\overline{RI} = []$        // list of replica-specific interleavings
5  forall $il \in ILs$ do
6      $indices \leftarrow index\_in\_interleaving(rId, il)$
7      if $indices \notin grouped\_by\_indices$ then
8          $grouped\_by\_indices[indices] = []$
9      $grouped\_by\_indices[indices].append(il)$
10 forall $(idx, il) \in grouped\_by\_indices$ do
11     $evs \leftarrow events\_after\_indices(il, idx)$
       /* remaining events to interleave at other
          replicas after $idx$ in the interleaving $il$      */
12     $grouped\_events.append(evs)$
13 $\overline{RI} = exclude(ILs, grouped\_events)$
   /* exclude replica-specific interleavings from
      existing interleavings list                        */
14 return $\overline{RI}$
```

3.4 Event Independence

Intuition: ER-π groups together events whose impact on replicated states has been determined to be independent. Considering their interleavings would be wasteful, as long as one can ascertain that no events between their occurrences have any impact on them. As ER-π starts executing the initially generated interleavings one by one, developers can apply this pruning algorithm dynamically by adding information about the newly discovered mutually independent events. Such dynamic pruning would be reasonable to apply in the case of a large number of initial interleavings that take a long time to replay.



ev_α
ev_β
ev_γ
. . .
$elem_\alpha$
$elem_\beta$
$elem_\gamma$
. . .
idx_α
idx_β
idx_γ
If Explored: {...ev_α, ev_β, ev_γ...}
Consider Explored: {...ev_α, ev_γ, ev_β...}
Consider Explored: {...ev_β, ev_α, ev_γ...}

Figure 5. Event Independence Pruning

Example: Consider a list data structure in Figure 5, events ev_α, ev_β, and ev_γ modify elements $elem_\alpha$, $elem_\beta$, and $elem_\gamma$, respectively. The list indices they modify are different, idx_α, idx_β, and idx_γ respectively. Consider that, by running several interleavings, a developer determines that these three events are executed independently having no impact on each other. Let's assume in an interleaving, the logical timestamps of these three events are t_α, t_β, and t_γ respectively and ($t_\alpha < t_\beta < t_\gamma$). Further, no other intermediate events can affect any of these three events. As a result, the interleavings, in which only the order of these three events changes (other events interleave in the same order) can be merged into a single interleaving. Hence, with this pruning logic, if there are k interleavings, each of which only the independent events interleave in a different order, ER-π considers the k interleavings as a single one, thus reducing the interleavings number by $k - 1$. For this above example, as there are three independent events, with this pruning logic, ER-π can reduce the number of interleavings by $3! - 1 = 5$. Algorithm 3 shows the step-by-step pseudo-code that applies event-independent-based pruning logic.

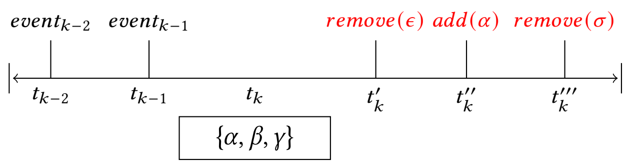
3.5 Failed Ops

Intuition: The constraints of some replicated data structures can cause some of their update operations to fail if they have been preceded by certain other update operations. Consider two replicas trying to add the same element to a replicated set; the constraints of the set data structure would

Algorithm 3: Event-Independence Pruning

```
Input: $ILs$: list of existing interleavings
Input: $IEvs$: list of independent events
Output: $\overline{EI}$: interleavings pruned by event-independency
$ILs \leftarrow read\_interleavings()$
$IEvs \leftarrow read\_independent\_events()$
$grouped\_interleavings = []$                // a list
$grouped\_by\_indices = \{\}$
/* a map, (key,value) ≡ (indices,interleaving)      */
$\overline{EI} = []$       // list of event-independent interleavings
forall $il \in ILs$ do
    $indices \leftarrow independent\_events\_indices(IEvs, il)$
    if $indices \notin grouped\_by\_indices$ then
        $grouped\_by\_indices[indices] = []$
    $grouped\_by\_indices[indices].append(il)$
forall $(idx, il) \in grouped\_by\_indices$ do
    $index\_first \leftarrow idx[0]$
    $index\_last \leftarrow idx[length(idx) - 1]$
    $Evs \leftarrow Events(il, index\_first, index\_last)$
    /* events to interleave during the first and last
       independent events in the interleaving il      */
    if $\forall ev \in Evs, \nexists iev \in IEvs : R(ev, iev)$ then
        /* in between events that don't impact on the
           independent events                         */
        $grouped\_interleavings.append(il)$
$\overline{EI} = exclude(ILs, grouped\_interleavings)$
/* exclude independent events-based interleavings from
   existing interleavings list                        */
return $\overline{EI}$
```

allow only one of them to succeed. As a result, if it can be determined that a certain conflicting update has been successfully executed, ER-π excludes the conflicting successor events from further consideration.



If Explored: $\{...remove(\epsilon), add(\alpha), remove(\sigma)...\}$

Consider Explored: $\{...remove(\epsilon), remove(\sigma), add(\alpha)...\}$

Consider Explored: $\{...add(\alpha), remove(\epsilon), remove(\sigma)...\}$

Figure 6. Failed Ops Pruning

Example: Consider the example of a Set in Figure 6. For a set, an event that tries to add an existing element or remove a non-existing element fails due to the set's constraints. At the timestamp, t_k, the set contains elements, α, β, and γ. So, with the set's content, the later events, $remove(\epsilon)$, $add(\alpha)$, and $remove(\sigma)$ at the timestamps t'_k, t''_k, and t'''_k respectively, become failed ops. As a result, in the interleavings, in which only these three events interleave in a different order, these interleavings can be merged. For the three events in this example, they can become failed ops and can interleave in a different order by 3! = 6 ways. Hence, considering them as a single interleaving reduces the number of interleavings by 5. Although small, this reduction further contributes to making the problem state more manageable. Besides, in some applications, the number of failed ops can go up significantly, thus increasing the impact of this pruning. Algorithm 4 shows the step-by-step pseudo-code that applies failed-ops-based pruning logic.

Algorithm 4: Failed-Ops Pruning

```
Input: $ILs$: list of existing interleavings
Input: $PEvs$: list of predecessor events
Input: $SEvs$: list of successor events
Output: $\overline{FI}$: interleavings pruned by failed-ops
$ILs \leftarrow read\_interleavings()$
$PEvs \leftarrow read\_predecessor\_events()$
$SEvs \leftarrow read\_successor\_events()$
$grouped\_by\_indices = \{\}$
/* a map, (key,value) ≡ (indices,interleaving)      */
$\overline{FI} = []$            // list of failed-ops interleavings
forall $il \in ILs$ do
    $pIdx \leftarrow predecessor\_events\_indices(PEvs, il)$
    $sIdx \leftarrow successor\_events\_indices(SEvs, il)$
    if $\forall p \in pIdx, \exists s \in sIdx : p \prec s$ & $\forall (p', p'') \in$
      $pIdx, \exists (s', s'') \in sIdx : p' \prec p'' \Rightarrow s' \prec s''$ then
        /* every predecessor event occurs before every
           successor event and the relative positions
           of these events remain the same            */
        if $concat(pIdx, sIdx) \notin grouped\_by\_indices$ then
            // group predecessor and successor events
            $grouped\_by\_indices[concat(pIdx, sIdx)] = []$
        $grouped\_by\_indices[concat(pIdx, sIdx)].append(il)$
$\overline{FI} = exclude(ILs, grouped\_by\_indices)$
/* exclude failed-ops-based interleavings from
   existing interleavings list                        */
return $\overline{FI}$
```

4 System Design and Workflow

We implement our approach as ER-π, an integration testing framework whose design is driven by the goals of providing a high degree of reusability and platform independence. It provides an exhaustive replaying middleware service for integration testing of replicated data systems. Figure 7 depicts the main components of ER-π along with the general workflow, which we describe in turn.

4.1 Proxying RDL Functions

ER-π detects and executes the possible interleavings of RDL events without modifying the library code by hand. To do so, ER-π proxies the RDL functions invoked from the application

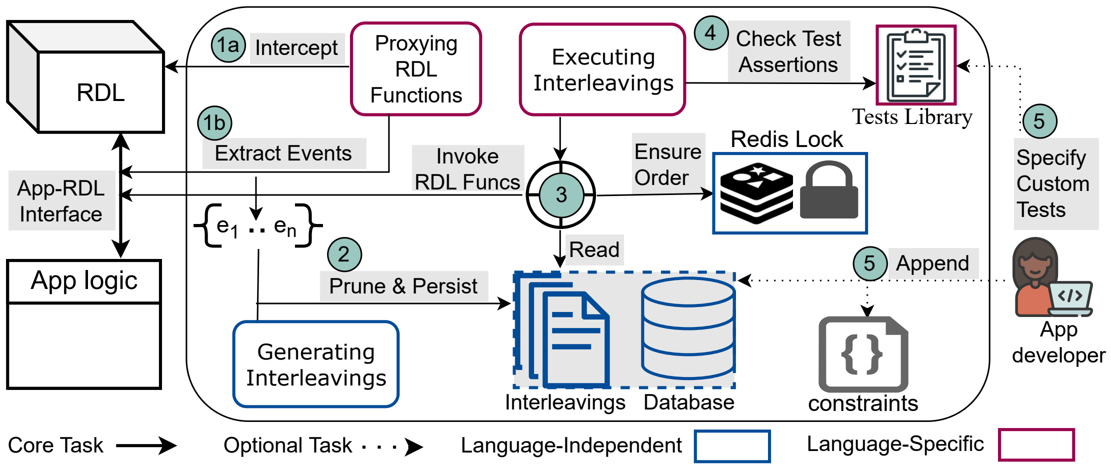


Figure 7. ER-π: Main Components and General Flow

code. The proxy techniques are language-specific and depend on the RDL programming languages. Proxies are provided via techniques that include Abstract Syntax Tree (AST) rewriting, Monkey patching, and Dynamic Proxy. We comment on these techniques in Section 5.1. ER-π provides higher-order functions, `ER-π.Start([...])` and `ER-π.End([...])` to designate the segment in the application logic for which the developers intend to conduct integration testing. ER-π intercepts which library functions have been invoked in the segment 1a , extracting them as events 1b to generate exhaustive interleavings between the `Start` and `End` points.

4.2 Generating Interleavings

This language-independent component of ER-π generates the possible interleavings to replay from the extracted events. While generating the interleavings, ER-π uses its event-grouping and replica-specific pruning algorithms to reduce the search space. However, the replica-specific algorithm is applied if the application developers want to examine particular replicas. In that case, ER-π allows specifying the replicas' id as a parameter of higher-order functions that ER-π is providing to mark the `Start` and `End` points. Upon generating the interleavings, ER-π assigns a Lamport timestamp [36] to each event in each interleaving. This Lamport timestamp defines event execution order during the replay. Having generated all possible interleavings, ER-π persists them in a database, 2 .

4.3 Executing Interleavings

To execute the generated interleavings 3 , ER-π relies on the higher-order `Start` and `End` functions to identify the execution boundary. Interleavings are read and replayed one by one. ER-π checkpoints the replicas' states and resets them prior to executing each interleaving, ensuring that interleaving executions remain unaffected by each other. ER-π invokes interleaving events via RDL proxies, enforcing the required event order via a distributed lock. The lock uses a Redis-provided distributed locking library [18]. This component deploys a mutex with a shared key managed by a Redis server, thus effecting the required distributed order in each interleaving.

4.4 Checking Test Assertions

After each interleaving, ER-π checks if any test assertions have been violated, 4 . ER-π provides a test library of commonly held wrong assumptions and misconceptions of RDL usage (discussed in Section 6.2). Provided as functions, the tests can be invoked after each interleaving. The following simplified Go code snippet demonstrates how to check if moving List items would not duplicate them in a replica.

```go
ER-π.Start()
........
copyList := replicaState.GetList()
moveItems(replicaState, fromIdx, toIdx)
........
ER-π.End(assertNoDuplication(copyList))

func assertNoDuplication(copyList []int) {
  assert.Equal(len(replicaState.GetList()), len(copyList))
  assert.True(reflect.DeepEqual(replicaState.GetList(),
    copyList), "Expected items to be equal")
}
```

4.5 Optional Features

ER-π also provides optional features that include the ability to specify custom test assertions and to add new event constraints 5 . Developers can specify a custom test as a function that can be passed as a parameter to `ER-π.End()`. When executing an integration test interactively, developers may discover some unique event constraints. These constraints can be passed as additional parameters to ER-π, which then further prunes the possible interleavings. ER-π's `event-independence` and `failed-ops` pruning algorithms are specifically designed to support such scenarios.

5 Implementation & Application

In this section, we describe the implementation of ER-π and its application to third-party replicated data systems, as an approach for testing the integration of RDLs.

5.1 Implementation

ER-π's language-independent components generate, prune, and persist interleavings, also controlling their event order. These components are shared across different languages. To serve as a reliable and efficient shared unit of functionality, these components must execute robustly across any platform or environment, supporting the diverse range of existing RDLs. C++ stands out for its support for heterogeneity, equipped with a mature compilation and execution infrastructure that ensures portability across platforms [14]. Thus, implementing these reusable components in C++ meets all key requirements for cross-platform compatibility. It took us $\approx 2K$ lines of C++ code to implement these language-agnostic components.

To further enhance language independence, ER-π manages interleavings in Datalog, a logic language known for its deductive storage capabilities that enable lightning-fast querying across large datasets [48]. ER-π initially stores the exhaustive set of $n!$ interleavings in Datalog's deductive database, using logic queries to perform the applicable pruning. ER-π generates the Souffle dialect of Datalog [20], with the code size varying based on the number of interleavings and pruning criteria.

One of ER-π's key design objectives is to make it possible to detect and replay the event interleavings without manual modification of RDLs' source code. To accomplish this objective, ER-π provides language-specific bindings, deliberately kept small in size and complexity. These bindings are used to generate proxies for the library functions, thus enabling ER-π to detect which RDL events are invoked during the specified workload. Later, ER-π reuses the proxied functions to replay the interleavings. With that goal, ER-π takes advantage of the language-specific techniques of the target RDLs. To demonstrate generality, we have applied ER-π to RDLs implemented in compiled (Go), interpreted (JavaScript), and managed (Java) languages. We describe the techniques used for each target language next.

5.1.1 Go. Go's standard library provides features for enhancing code with additional functionality, without requiring any manual modification. Specifically, we use `go/ast`, which interfaces with the Go compiler to expose an Abstract-Syntax Tree (AST) [52]. By modifying AST, we introduce the needed proxy generation functionality in fewer than 300 lines of Go code.

5.1.2 JavaScript. Interpreted languages offer a high degree of adaptability. Using Monkey Patching, one can proxy any JavaScript function [33]. Proxied functions can be intercepted and customized. This feature makes our JavaScript bindings particularly concise, so all needed methods take fewer than 280 lines of code to proxify.

5.1.3 Java. A managed language, Java is compiled to bytecode, which is executed by a virtual machine. Bytecode is easily amenable to modification, both statically and dynamically [15]. The Java virtual machine provides the Dynamic Proxy [22] facility that generates bytecode at runtime to proxify any interface method. This feature enables us to proxify all required methods in our target RDL in fewer than 415 lines of Java code.

5.2 ER-π in Action

Procedure Workflow

```
...............
ER-π.Start()
State 1:
if first run then
    events ← extract_events()      // by proxy functions
    algos ← applicable_pruning_algorithms(events)
State 2:
$\overline{ILs}$ ← generate_interleavings(events, algos)
persist($\overline{ILs}$)
State 3:
forall il ∈ $\overline{ILs}$ do
    initS ← replicaStates()  // replicas' initial states
    execute(il)
    ER-π.InvokeTests(...)
    /* invoke built-in and custom tests            */
    reset(initS)              // reset to initial states
State 4:
if new constraints then
    algos ← suitable_pruning_algorithms()
    go to State 2
ER-π.End()
..............
```

Applying ER-π to third-party replicated data systems also proved manageable for us from a configuration standpoint. It only required adding several instructions to the existing build and configuration scripts. ER-π's language-agnostic components rely on C++ and Souffle, which have to be preinstalled. Recall that to mark the `Start` and `End` points of the workload for which the distributed events are to be detected and interleaved, ER-π provides higher-order functions for developers to parameterize. The language-agnostic components of ER-π comprise its runtime that receives the events to interleave, subsequently pruning and persisting the interleavings. Also, ER-π's system design allows application developers to provide constraints for further pruning at runtime. ER-π periodically checks for the presence of JSON files in the `constraints` directory. If found, ER-π then consults the files for the new constraints to apply, thus further reducing the problem space. Procedure Workflow demonstrates ER-π's major steps and their connections, once it has been applied to conduct RDL integration testing.

6 Evaluation

The evaluation of ER-π is guided by the following questions:

1. **RQ1: Reproducing Bugs:** How effective is ER-π to reproduce RDL integration bugs?
2. **RQ2: Recognizing Misconceptions:** Is ER-π's exhaustive replay a viable means of recognizing common RDL integration misconceptions?
3. **RQ3: Reducing Problem Space:** How effective are ER-π's pruning algorithms at reducing the problem space of exhaustive replay?

We applied ER-π to five third-party replicated data systems, implemented in Go, Java, and JavaScript. Next, we briefly describe our evaluation subjects.

Subject 1: Soundcloud's Roshi. Roshi provides a time-series event database using a Last-Write-Win (LWW) CRDT semantics for conflict resolution [13]. Implemented in Go, Roshi adds a stateless, distributed layer on top of Redis and stores shared dataset copies in multiple independent Redis instances, mainly used for SoundCloud's streaming. For ER-π's evaluation, we prototyped the application logic that invokes Roshi's functions to maintain the replicated state.

Subject 2: OrbitDB. OrbitDB is another database, whose properties include being serverless, peer-to-peer, and eventually consistent by means of Merkle-CRDTs [42] for conflict-free writes and merges [59]. OrbitDB is implemented in JavaScript. For ER-π's evaluation, we used OrbitDB by prototyping application logic that invokes its RDL functions.

Subject 3: ReplicaDB. ReplicaDB, another open-source database, relies on an RDL to transfer data in bulk between relational and NoSQL representations [41]. Implemented in Java, ReplicaDB can operate across environments, providing different replication modes with parallel data transfer. For ER-π's evaluation, we prototyped application code that invokes ReplicaDB's functions to maintain its replicated state.

Subject 4: Yorkie. Yorkie is a Go-based replicated document store whose JSON-represented documents allow for seamless collaboration by means of CRDT [23]. For ER-π's evaluation, we used Yorkie by prototyping application logic that invokes its RDL functions.

Subject 5: CRDTs. CRDTs is a collection of RDL data structures implemented in Java [25]. For ER-π's evaluation, we used CRDTs by adding application logic on top of it that invokes its RDL functions to maintain its replicated state.

Experimental Setup. For our experimental evaluation, we used a virtual distributed environment that we configured to have three replicas. Evaluating against three replicas has been a common strategy for conducting research experiments, concerned not only with replicated data systems [43] but also with numerous cloud databases and edge computing systems [53]. Two replicas were hosted on 64-bit Ubuntu 20.04 laptops, the first with 32 GB RAM and an Intel Core i7 Processor and the second with 8 GB RAM and an Intel Core i5 Processor. The third replica was hosted on a 32-bit Raspbian 9 Raspberry Pi 3 with 1 GB RAM and ARMv7 Quad Core Processor.

6.1 RQ1: Reproducing Bugs

One possible application of ER-π is to reproduce the bugs experienced in a deployed system on the developer's end. When a bug is experienced during the execution of a replicated data system, it might be impossible for users to report which of the possible interleavings was in effect when the bug manifested itself. To evaluate ER-π's effectiveness for this task, we reproduced several previously reported bugs in our evaluation subjects, as listed in Table 1. Because many of these bugs have already been fixed, oftentimes we had to use a previous release of our subjects with the bug still in the codebase.

BugName	Issue#	#Events	Status	Reason
Roshi-1 [2]	18	9	closed	misconception
Roshi-2 [1]	11	10	closed	RDL issue
Roshi-3 [3]	40	21	closed	misconception
OrbitDB-1 [5]	513	12	open	—
OrbitDB-2 [4]	512	8	open	—
OrbitDB-3 [12]	1153	15	closed	misuse
OrbitDB-4 [7]	583	18	closed	misconception
OrbitDB-5 [6]	557	24	closed	misconception
ReplicaDB-1 [9]	79	10	closed	misuse
ReplicaDB-2 [8]	23	14	closed	misconception
Yorkie-1 [11]	676	17	open	—
Yorkie-2 [10]	663	22	closed	misconception

Table 1. Bug benchmarks. "`#Events`"—# of interleaved events. "`Status`"—if the bug is closed by library developers. "`Reason`"—what causes the bug.

6.2 RQ2: Recognizing Misconceptions

Even though RDLs are intended to facilitate the implementation of replicated data systems, developers might hold incorrect assumptions about how RDLs interact with the application logic. Such assumptions, referred to as *misconceptions*, can cause RDL usage mistakes, leading to bugs and vulnerabilities. Five common RDL misconceptions are:

#1 The underlying network ensures causal delivery [56].
#2 The order of List elements is always consistent [56].
#3 Moving items in a List doesn't cause duplication [24].

#4 Sequential IDs are always suitable for creating new items in a to-do list [24].
#5 Multiple replicas in different regions mathematically resolve to the same state without coordination [26, 30].

For our evaluation, we pursued a strategy similar to seeding bugs, commonly used for assessing the effectiveness of bug-detecting tools. However, "seeding misconceptions" required devising more elaborate strategies than those used for seeding simple bugs. For each of these five misconceptions, we next describe both our seeding strategy and the test procedure used for detecting it.

Misconception #1: To seed #1, we stopped invoking the conflict-resolution function for a particular replica and randomly interleaved the same set of events requested from other replicas. By assuming that #1 is correct, the programmer can expect different orders of requested events to be causally sorted, thus always bringing the replica to the same state. To detect this misconception, we wrote a test that compares the replica's states, which resulted from different interleavings. Running this test revealed that without invoking the conflict-resolution function explicitly, rather than just depending on the underlying network, the replica's state diverges from one interleaving to another. Indeed, one cannot rely on the network for sorting the causal updates, with the responsibility falling on the consistency protocol.

Misconception #2: To seed #2, we used a replicated list data structure, with its elements left unsorted. To detect the misconception, we checked the order of list elements in different replicas in multiple interleavings and observed the order varying between replicas. Even in the same replica, depending on the execution order of updates (add or remove elements), different interleavings can cause dissimilar element orderings.

Misconception #3: To seed #3, we implemented a typical `move` operation that applies a `delete` operation followed by an `insert` operation. The potential problem is that if no special care is taken, the replicated state can contain duplicated elements. To ensure the correctness of concurrently moving the same element to different positions across replicas, one must designate a particular position as "winning" [29]. To detect this misconception, we interleaved the move of the same element to different positions across replicas and checked if any duplication was present.

Misconception #4: To seed #4, we implemented a to-do list using sequential IDs for each to-do item. To create a new item, our implementation generates an ID by incrementing the highest ID. The potential problem is that concurrently creating to-do items in different replicas can cause a clash. For example, if two replicas see the highest ID as n, they both will create new to-do items with the same ID of $n + 1$. One strategy to avoid this problem is suggested by AMC [24]: add consistent initialization, use a random number generator for IDs, and ensure that to-dos are added to the same map. To detect this misconception, we interleaved the event of creating new to-do items across multiple replicas, checking for any clashes across IDs in their synchronization requests.

Misconception #5: To seed #5, we prototyped the scenario described by our motivating example in Section 2.3. Specifically, we stopped coordination with other replicas for a particular replica. To ensure correctness, each replica must coordinate with the remaining replicas to reflect each other's updates. To detect this misconception, we tested whether the particular replica's state results in dissimilar states across different interleavings.

Subjects	Misconception label#				
	#1	#2	#3	#4	#5
Roshi	✓	✓	✓		✓
OrbitDB	✓				✓
ReplicaDB	✓				
Yorkie	✓				✓
CRDTs	✓	✓	✓	✓	✓

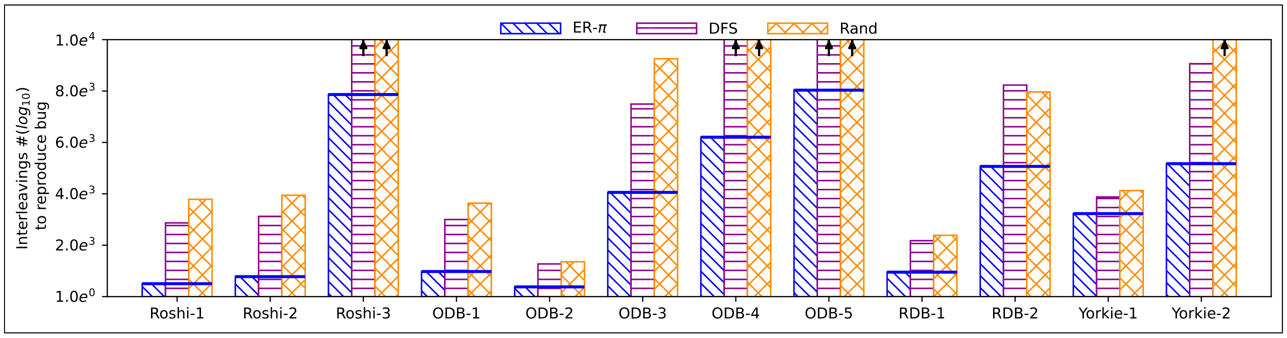
Table 2. Recognizing Misconceptions with ER-π

Table 2 depicts the misconceptions ER-π manages to detect in the evaluation subjects.

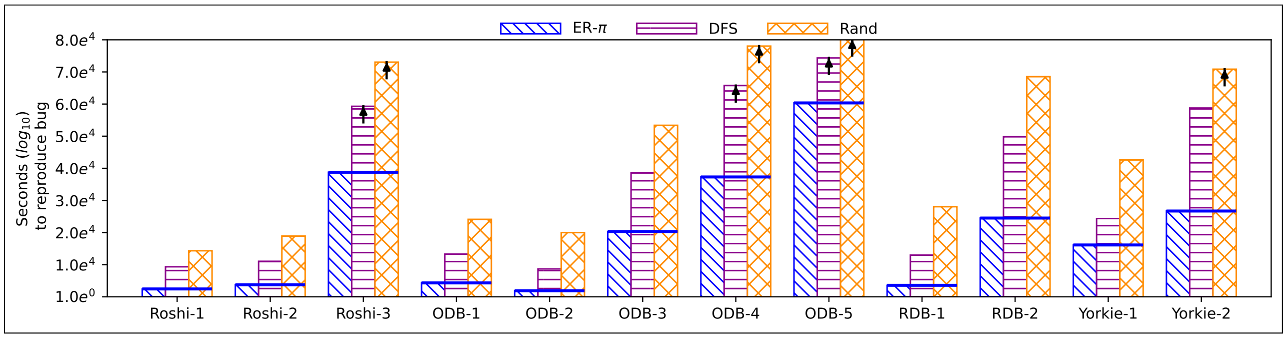
6.3 RQ3: Reducing Problem Space

To evaluate how ER-π's pruning algorithms reduce the problem space, we reproduced each bug in Table 1, measuring the number of interleavings and time required for reproduction. The reproduction entailed replaying the interleavings in three modes; (1) ER-π with its applicable pruning algorithms, (2) `DFS`, and (3) Random (`Rand`). For n events, both DFS and Rand exhaustively explore all $n!$ interleavings. These modes, however, order the interleavings differently. DFS treats the interleavings as a tree that starts at an empty root node and recursively explores each event, with each level branching out to unvisited events by backtracking and expanding. Every path from root to leaf represents a unique interleaving, capturing all possible $n!$ interleavings. In contrast, Rand composes each interleaving by randomly shuffling the events, caching the composed interleavings to avoid repetition. All experiments generated interleavings at runtime for all three modes. Overall, the evaluation explored $10K$ interleavings for each bug, taking us around seven machine days to complete the entire experiment.

Figure 8a depicts the number of interleavings (in log_{10} scale) it took to reproduce each bug. If exploring $10K$ interleavings fails to reproduce a bug, its corresponding bar is marked with $\uparrow$. Both DFS and Rand failed to reproduce the bugs Roshi-3, OrbitDB-4, and OrbitDB-5, while Rand also failed to reproduce the bug Yorkie-2. In contrast, ER-π reproduced each bug, taking at most around $8K$ interleavings for the bugs Roshi-3 and OrbitDB-5. DFS outperformed Rand,



(a) Number of Interleavings Explored to Reproduce Bug



(b) Time Required to Reproduce Bug

Figure 8. Number of interleavings and time (both in log_{10} scale) required to reproduce bugs. ODB and RDB stands for OrbitDB and ReplicaDB respectively. ↑ indicates that the bug is not reproduced after exploring 10, 000 interleavings.

except for ReplicaDB-2. The non-deterministic nature of random exploration sometimes leads to unexpected outcomes, by chance, discovering this bug faster.

Figure 8b depicts the reproduction time (also in log_{10} scale) for each bug. Because we terminated the experiment after exploring $10K$ interleavings having failed to find a bug, the figure shows the time elapsed before the termination, similarly marked with ↑. Replaying $10K$ interleavings uninterruptedly took as long as $80K$ seconds (almost one machine day), explaining the need for our termination threshold. For all bugs, Rand took the most time due to the need to keep shuffling the events until finding an unexplored interleaving. For this reason, sometimes, small differences between the number of interleavings translate into much larger time differentials, such as for the bugs OrbitDB-2, ReplicaDB-1, and Yorkie-1.

ER-π's faster reproduction is due to its pruning algorithms that reduce the problem space. Figure 9 shows how each algorithm contributes to reducing the number of interleavings. Compared to DFS and Rand, ER-π prunes $\approx 5.6\times$ and $\approx 7.4\times$ interleavings to replay on average, thus reducing the time to reproduce a bug by $\approx 2.78\times$ and $\approx 4.38\times$, respectively.

To evaluate ER-π's scalability, we designed a microbenchmark around the bug OrbitDB-5. Rather than terminating the execution after $10K$ interleavings, we left the experiment running. Figure 10 shows five experimental runs with the

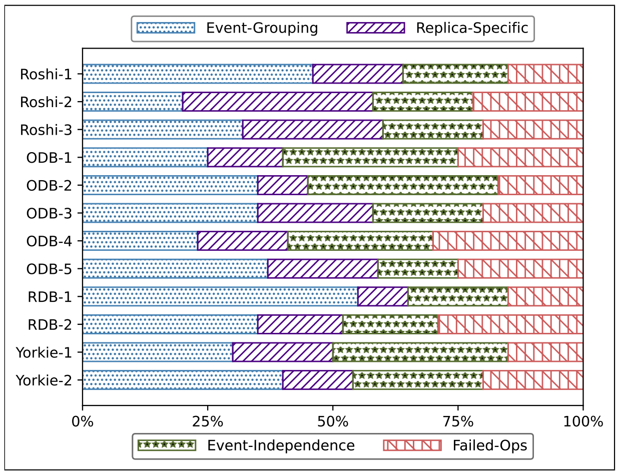


Figure 9. Individual Algorithm's Contribution to the Reduction of Interleavings Number

same setup, modes, and events. Each run either (✓): successfully reproduced the bug or (×): exhausted all allocated resources, causing the system to crash before reproducing the bug. ER-π successfully reproduced the bug for all runs. In contrast, the competing modes crashed before reproducing the bug, except for one successful DFS run.

We configured the experiment to terminate upon reproducing the bug. Since ER-π consistently succeeded, it never

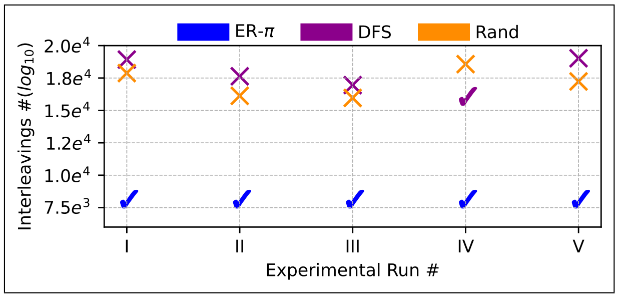


Figure 10. "Succeed-or-Crash" Micro-Benchmark

reached the point of resource exhaustion. In contrast, the observed crashes in other approaches were due to exhausting allocated system resources. While our findings are inherently setup-specific, they underscore ER-π's ability to enhance test scalability. For the same number of distributed events, ER-π effectively reduces the problem search space by minimizing the number of interleavings to replay. As a result, it scales to a higher number of distributed events than the evaluated competitors, which lack pruning capability.

Applicability and Limitations: ER-π can be applied to any replicated data system that integrates an RDL. Since our work focuses on integration testing, the system under test must comprise distinct components representing business logic and replicated data management. With the widespread adoption of component-based design—driven by its benefits in modularity and overall software quality [49]—ER-π is well-suited for modern distributed software ecosystems, making it broadly applicable across existing systems.

Our approach has certain limitations, stemming from our target domain and specific engineering choices. ER-π's pruning algorithms are inherently domain-specific, designed specifically for eventually consistent RDLs. As they stand, these algorithms cannot be directly applied to other domains. They may be adaptable to other distributed computing constraints—an avenue we have yet to explore. Additionally, ER-π interfaces with RDL code through proxying rather than code instrumentation. As a result, its applicability may be limited in language ecosystems that lack robust proxying support. Furthermore, its runtime pruning relies on developer-provided constraints rather than automatic inference via dynamic analysis. Despite its domain-specific focus and design choices, ER-π's main value lies in providing a versatile infrastructure for capturing and replaying interleavings, while applying powerful optimizations that reduce the problem search space. Consequently, ER-π proves particularly effective in ensuring the robustness and reliability of replicated data systems that integrate third-party RDLs.

7 Related Work

The design of ER-π draws inspiration from several research areas, including the model checking of distributed systems and testing replicated data systems, which we discuss below.

Model Checking of Distributed Systems. Model checking explores interleavings of non-deterministic events to uncover hard-to-find bugs. To be effective, checkers must be able to reduce the exploration path of the interleaved events. FlyMC introduces three path-reduction algorithms that provide high scalability and fast performance for data center systems by efficiently reducing the problem space [35]. DPOR presents a partial-order reduction algorithm based on an initially explored arbitrary interleaving of the concurrent local threads and then dynamically tracking interactions between them to identify alternative paths [21]. MODIST interleaves a variety of network conditions and failures, such as message reordering, network partitions, and machine crashes in distributed systems, providing a customizable framework, whose strategies include DPOR, random exploration, and DFS [55]. SAMC, a distributed model checker for cloud systems based on a white-box principle, takes the target system's simple semantic information and incorporates it into four novel reduction policies to scalably find deep bugs [32]. MACEMC combines DFS and random path-exploration techniques to find liveness bugs in system code, with testers manually prioritizing events [28]. dBug systematically explores possible orders of distributed and concurrent events for a given workload by adopting partial-order reduction to produce a trace for replaying error encounters [47]. CrystallBall's state exploration algorithm adopts dynamic path-reduction to explore causally related distributed events for detecting any property violation in a black-box manner [54]. These related works target either certain properties of the system or network messages across the nodes as a model, without considering replication or eventual consistency. In contrast, ER-π targets the interfacing of RDL with application code as a model to identify integration violations, while also relying on exhaustive replay of possible interleavings.

Testing RDL. With the increasing availability of powerful third-party RDLs, testing them has become an active research topic. MET presents a design and implementation-level verification framework for CRDT and applies explorative testing by permuting all nondeterministic message orderings [56]. AMC deterministically explores the behavior of the JSON CRDT library, Automerge, and attempts to cover both common and edge behavior along with the implementation [24]. VeriFx provides a framework for describing the semantics of RDL at a high level and checking the properties of the model. Additionally, it can synthesize the implementations of the RDL into executable code, thus presenting an approach, useful for creating new RDL [16]. Katara synthesizes verified CRDT designs from sequential data type implementations, trying to eliminate errors by design verification [31]. Compared to these works, which target the design and implementation verification of RDLs themselves, ER-π specifically focuses on testing the integration of RDL with the application code. Although the design of ER-π draws inspiration from these prior state-of-the-art approaches and tools, ER-π's unique contribution lies in its focus on integrating testing.

8 Conclusion

We have presented ER-π, a middleware framework for integration testing of replicated data systems. By identifying and replaying the possible interleavings of RDL events, ER-π has been shown capable of finding difficult-to-find bugs and RDL usage misconceptions. ER-π's novel four pruning algorithms effectively reduce the problem space. ER-π's middleware design enables applying it to RDLs written in different languages, without manually modifying their source code. To the best of our knowledge, ER-π offers the first instance of exhaustive interleaving replay designed specifically for testing RDL integration.

As our future work directions, we plan to further expand the scope of our evaluation to other replicated data systems. We plan to extend the applicability and usefulness of ER-π for tasks such as resource profiling and fuzzing. As integration testing is fundamental to ensure the correctness of interfaces between application and third-party middleware libraries, we plan to explore how our approach can help improve the robustness of large-scale replicated data systems. Finally, we plan to explore whether the lessons learned from this work can be applied to other distributed computing domains to enhance their verification and testing.

Acknowledgments

We sincerely thank our shepherd Mohammad Sadoghi and the anonymous reviewers, whose feedback has helped improve this paper. This research was supported by NSF through Grant #2232565.

References

[1] 2014. BUG: Roshi-11: CRDT semantics violated if same timestamp? https://github.com/soundcloud/roshi/issues/11.
[2] 2014. BUG: Roshi-18: Incorrect deleted field in response. https://github.com/soundcloud/roshi/issues/18.
[3] 2015. BUG: Roshi-40: roshi-server golang app select and map order? https://github.com/soundcloud/roshi/issues/40.
[4] 2018. BUG: OrbitDB-512: Lamport clock can be set far into future making db progress halt. https://github.com/orbitdb/orbitdb/issues/512.
[5] 2018. BUG: OrbitDB-513: Ordering tie breaker can cause undefined ordering with the same identity. https://github.com/orbitdb/orbitdb/issues/513.
[6] 2019. BUG: OrbitDB-557: repo folder keeps getting locked. https://github.com/orbitdb/orbitdb/issues/557.
[7] 2019. BUG: OrbitDB-583: Head hash didn't match the contents errors. https://github.com/orbitdb/orbitdb/issues/583.
[8] 2021. BUG: ReplicaDB-23: deleted records aren't getting deleted from the sink tables. https://github.com/osalvador/ReplicaDB/issues/23.
[9] 2022. BUG: ReplicaDB-79: Out of memory error. https://github.com/osalvador/ReplicaDB/issues/79.
[10] 2023. BUG: Yorkie-663: Modify the set operation to handle nested object values. https://github.com/yorkie-team/yorkie/issues/663.
[11] 2023. BUG: Yorkie-676: Document doesn't converge when using Array.MoveAfter. https://github.com/yorkie-team/yorkie/issues/676.
[12] 2024. BUG: OrbitDB-1153: Uncaught Error: Could not append entry: although write access is granted. https://github.com/orbitdb/orbitdb/issues/1153.
[13] Peter Bourgon and Nick Stenning. 2014. Roshi. https://github.com/soundcloud/roshi.
[14] Gordon Brown, Ruyman Reyes, and Michael Wong. 2019. Towards Heterogeneous and Distributed Computing in C++. In *Proceedings of the International Workshop on OpenCL*. 1–5.
[15] Giacomo Cabri, Luca Ferrari, and Letizia Leonardi. 2005. Injecting Roles in Java Agents Through Runtime Bytecode Manipulation. *IBM Systems Journal* 44, 1 (2005), 185–208.
[16] Kevin De Porre, Carla Ferreira, and Elisa Gonzalez Boix. 2022. VeriFx: Correct Replicated Data Types for the Masses. *arXiv preprint arXiv:2207.02502* (2022).
[17] Kevin De Porre, Carla Ferreira, Nuno Preguiça, and Elisa Gonzalez Boix. 2021. ECROs: Building Global Scale Systems from Sequential Code. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–30.
[18] Redis Documentation. 2023. Distributed Locks with Redis. https://redis.io/docs/latest/develop/use/patterns/distributed-locks/.
[19] Mostafa Elhemali, Niall Gallagher, Bin Tang, Nick Gordon, Hao Huang, Haibo Chen, Joseph Idziorek, Mengtian Wang, Richard Krog, Zongpeng Zhu, et al. 2022. Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 1037–1048.
[20] Zhiwei Fan, Sunil Mallireddy, and Paraschos Koutris. 2022. Towards Better Understanding of the Performance and Design of Datalog Systems. *Datalog* 2 (2022), 166–180.
[21] Cormac Flanagan and Patrice Godefroid. 2005. Dynamic Partial-Order Reduction for Model Checking Software. *ACM Sigplan Notices* 40, 1 (2005), 110–121.
[22] George Fourtounis, George Kastrinis, and Yannis Smaragdakis. 2018. Static Analysis of Java Dynamic Proxies. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 209–220.
[23] Youngteac Hong and Dongcheol Choe. 2020. Yorkie. https://github.com/yorkie-team/yorkie.
[24] Andrew Jeffery and Richard Mortier. 2023. AMC: Towards Trustworthy and Explorable CRDT Applications with the Automerge Model Checker. In *Proceedings of the 10th Workshop on Principles and Practice of Consistency for Distributed Data*. 44–50.
[25] Andrejs Jermakovics. 2014. CRDTs. https://github.com/ajermakovics/crdts.
[26] Leena Joshi. 2022. How to simplify distributed app development with CRDTs. Accessed 2024.
[27] Gowtham Kaki, Swarn Priya, KC Sivaramakrishnan, and Suresh Jagannathan. 2019. Mergeable Replicated Data Types. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.
[28] Charles Killian, James W Anderson, Ranjit Jhala, and Amin Vahdat. 2007. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. NSDI.
[29] Martin Kleppmann. 2020. Moving Elements in List CRDTs. In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*. 1–6.
[30] Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, Natacha Crooks, and Joseph M Hellerstein. 2022. Keep CALM and CRDT On.

arXiv preprint arXiv:2210.12605 (2022).

[31] Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, and Joseph M Hellerstein. 2022. Katara: Synthesizing CRDTs with Verified Lifting. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 1349–1377.

[32] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F Lukman, and Haryadi S Gunawi. 2014. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 399–414.

[33] Benjamin S Lerner, Herman Venter, and Dan Grossman. 2010. Supporting Dynamic, Third-Party Code Customizations in JavaScript Using Aspects. *ACM Sigplan Notices* 45, 10 (2010), 361–376.

[34] Che-Sheng Lin and Gwan-Hwan Hwang. 2013. State-cover Testing for Nondeterministic Terminating Concurrent Programs with an Infinite Number of Synchronization Sequences. *Science of Computer Programming* 78, 9 (2013), 1294–1323.

[35] Jeffrey F Lukman, Huan Ke, Cesar A Stuardo, Riza O Suminto, Daniar H Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, et al. 2019. FlyMC: Highly Scalable Testing of Complex Interleavings in Distributed Systems. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16.

[36] Giridhar Manepalli. 2022. Clocks and Causality - Ordering Events in Distributed Systems. https://www.exhypothesi.com/clocks-and-causality/.

[37] David Mealha, Nuno Preguiça, Maria Cecilia Gomes, and João Leitão. 2019. Data Replication on the Cloud/Edge. In *Proceedings of the 6th Workshop on Principles and Practice of Consistency for Distributed Data*. 1–7.

[38] Ruijie Meng, George Pîrlea, Abhik Roychoudhury, and Ilya Sergey. 2023. Greybox Fuzzing of Distributed Systems. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 1615–1629.

[39] Anthony Russo and Joe Russo. 2018. Avengers: Infinity War. Walt Disney Studios Motion Pictures.

[40] Swarup Kumar Sahoo, John Criswell, and Vikram Adve. 2010. An Empirical Study of Reported Bugs in Server Software with Implications for Automated Bug Diagnosis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 485–494.

[41] Oscar Salvador and Francesco Zanti. 2018. ReplicaDB. https://github.com/osalvador/ReplicaDB.

[42] Hector Sanjuan, Samuli Poyhtari, Pedro Teixeira, and Ioannis Psaras. 2020. Merkle-CRDTs: Merkle-DAGs meet CRDTs. *arXiv preprint arXiv:2004.00107* (2020).

[43] Nazmus Saquib, Chandra Krintz, and Rich Wolski. 2022. Log-Based CRDT for Edge Applications. In *2022 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 126–137.

[44] Ohad Shacham, Mooly Sagiv, and Assaf Schuster. 2005. Scaling Model Checking of Dataraces Using Dynamic Information. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. 107–118.

[45] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free Replicated Data Types. In *13th International Conference on Stabilization, Safety, and Security of Distributed Systems (SSS 2011)*. Springer LNCS volume 6976, 386–400. https://doi.org/10.1007/978-3-642-24550-3_29

[46] Dharma Shukla. 2017. A Technical Overview of Azure Cosmos DB. https://azure.microsoft.com/en-us/blog/a-technical-overview-of-azure-cosmos-db/.

[47] Jiri Simsa, Randy Bryant, and Garth Gibson. 2010. dBug: Systematic Evaluation of Distributed Systems. In *5th International Workshop on Systems Software Verification (SSV 10)*.

[48] Yannis Smaragdakis and Martin Bravenboer. 2010. Using Datalog for Fast and Easy Program Analysis. In *International Datalog 2.0 Workshop*. Springer, 245–251.

[49] Tassio Vale, Ivica Crnkovic, Eduardo Santana De Almeida, Paulo Anselmo da Mota Silveira Neto, Yguaratã Cerqueira Cavalcanti, and Silvio Romero de Lemos Meira. 2016. Twenty-eight years of component-based software engineering. *Journal of Systems and Software* 111 (2016), 128–148.

[50] Dong Wang, Wensheng Dou, Yu Gao, Chenao Wu, Jun Wei, and Tao Huang. 2023. Model Checking Guided Testing for Distributed Systems. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 127–143.

[51] Feng Wang, Jie Qiu, Jie Yang, Bo Dong, Xinhui Li, and Ying Li. 2009. Hadoop High Availability through Metadata Replication. In *Proceedings of the first international workshop on Cloud data management*. 37–44.

[52] Adam Welc. 2021. Automated Code Transformation for Context Propagation in Go. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1242–1252.

[53] Rich Wolski, Chandra Krintz, Fatih Bakir, Gareth George, and Wei-Tsung Lin. 2019. CSPOT: Portable, Multi-scale Functions-as-a-Service for IoT. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*. 236–249.

[54] Maysam Yabandeh, Nikola Knežević, Dejan Kostić, and Viktor Kuncak. 2010. Predicting and Preventing Inconsistencies in Deployed Distributed Systems. *ACM Transactions on Computer Systems (TOCS)* 28, 1 (2010), 1–49.

[55] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI'09*. 213–228.

[56] Yuqi Zhang, Yu Huang, Hengfeng Wei, and Xiaoxing Ma. 2022. MET: Model Checking-Driven Explorative Testing of CRDT Designs and Implementations. *arXiv preprint arXiv:2204.14129* (2022).

[57] Yicheng Zhang, Matthew Weidner, and Heather Miller. 2023. Programmer Experience When Using CRDTs to Build Collaborative Webapps: Initial Insights. In *13th annual workshop on the intersection of HCI and PL (PLATEAU)*. https://doi.org/10.1184/R1/22277341.v1

[58] Xin Zhao and Philipp Haller. 2020. Replicated Data Types that Unify Eventual Consistency and Observable Atomic Consistency. *Journal of logical and algebraic methods in programming* 114 (2020), 100561.

[59] Friedel Ziegelmayer and Hayden Young. 2018. OrbitDB. https://github.com/orbitdb/orbitdb.