Toward Thorough and Practical Integration Testing of Replicated Data Systems

Provakar Mondal
Software Innovations Lab, Virginia Tech
Blacksburg, Virginia, USA
provakar@cs.vt.edu

ABSTRACT

Highly available applications rely on replicated data, but complex event interleavings between application logic and replicated data libraries (RDLs) often cause subtle integration bugs. Detecting such bugs is challenging due to the inherent nondeterminism of distributed execution, as certain bugs can only manifest under specific interleavings. Correctness testing, therefore, requires replaying all possible interleavings—a challenging task due to the combinatorial explosion of the interleaving space. My doctoral dissertation addresses this challenge with ER-π, a middleware framework that exercises all possible interleavings between the application code and RDL; it also eliminates redundant and impossible interleavings via novel pruning techniques. Initial results show that ER-π successfully reproduces 12 real-world bugs across multiple open-source RDLs while significantly reducing the interleaving search space. Our ongoing work extends this foundation with interleaving prioritization, ranking interleavings execution by their likelihood of exposing faults—particularly those introduced by recent code changes, thus accelerating bug discovery. This research supports developers responsible for ensuring the correctness and reliability of replicated data systems.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Replicated Data System, Integration Testing, Middleware Service, Problem Space Reduction, Test Prioritization

ACM Reference Format:
Provakar Mondal. 2025. Toward Thorough and Practical Integration Testing of Replicated Data Systems. In *26th International Middleware Conference (MIDDLEWARE Demos Posters and Doctoral Symposium '25), December 15–19, 2025, Nashville, TN, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3721464.3777426

1 INTRODUCTION

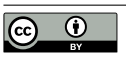
With the growing demand for replicated data systems, developers increasingly rely on expressive and intuitive third-party libraries that provide data replication support and robust conflict resolution [4]. These replicated data libraries (RDLs) allow developers to focus on application-specific business logic while offloading the complexity of managing replicated data. Application code interacts with RDLs, thus ensuring availability and consistency across replicas [10]. Subtle bugs can arise not from the individual components themselves but from their interactions under specific event interleavings [12]. Hence, an exhaustive integration testing that can check all possible event interleavings is essential to verify that the interactions between the application code and RDLs are correct.

Nevertheless, testing distributed applications is inherently difficult [19]. The non-deterministic execution of distributed systems leads to numerous possible interleavings of events, and certain bugs, such as destructive data races and consistent with incorrect results, can manifest only under specific interleavings [17]. Detecting and reproducing these bugs requires the ability to generate and replay all possible interleavings systematically. However, two challenges make this task particularly demanding. First, identifying potential interleavings is complex, especially where system components interact both locally and remotely. Second, the number of interleavings grows exponentially, rendering exhaustive exploration infeasible without sophisticated reduction techniques.

My dissertation research aims to make integration testing of replicated data systems both thorough and practical. We have created ER-π (**E**xhaustive **R**eplay of **P**ossible **I**nterleavings) [11], a middleware that enables integration testing for RDL-based applications. ER-π detects distributed events raised from the interaction between application code and RDLs, generates all possible interleavings, and applies four novel pruning algorithms to reduce redundant (covered by other interleavings) and impossible ones. It then replays each interleaving to help developers reproduce bugs. Applied to several open-source applications, ER-π successfully reproduces previously reported 12 bugs while keeping the problem space feasible to test exhaustively, demonstrating its integration testing effectiveness.

Building on the foundation of ER-π, my ongoing research focuses on prioritizing interleavings. While pruning reduces the search space, the remaining set of interleavings can still be prohibitively large. Prioritization can make testing more efficient by ranking interleavings according to their likelihood of exposing faults, particularly those introduced by recent code changes. My dissertation research, through a two-thrust approach that combines systematic pruning and prioritization of event interleavings, aims to advance integration testing of replicated data systems. This research makes the following contributions:

(1) ER-π, a middleware service that detects distributed events, generates all possible event interleavings, prunes and replays



MIDDLEWARE Demos Posters and Doctoral Symposium '25, December 15–19, 2025, Nashville, TN, USA

ACM ISBN 979-8-4007-1556-3/2025/12.
https://doi.org/10.1145/3721464.3777426

them for systematic integration testing between application code and RDLs (§ 2).

(2) Discussion of ongoing research that prioritizes interleavings according to their likelihood of revealing bugs, particularly those introduced by recent code changes, thereby improving testing efficiency (§ 3).

2 COMPLETED WORK

In this section, we describe the main contributions delivered so far, including ER-π's system design, workflow, and initial results.

2.1 ER-π

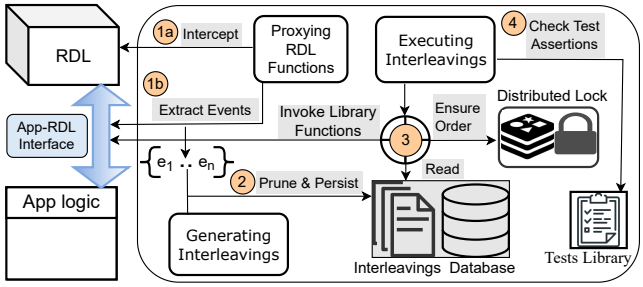


Figure 1: Pruning and Replaying Interleavings with ER-π

ER-π detects, prunes, and exhaustively replays interleavings between the interface of application code and RDLs. Figure 1 illustrates the main components of ER-π and its workflow. ER-π detects and executes the possible interleavings of RDL events by proxying the RDL functions invoked from the application code. Developers mark the boundaries of the workload to be tested using ER-π's provided higher-order functions `ER-π.Start([...])` and `ER-π.End([...])`. ER-π intercepts which RDL functions have been invoked in the workload 1a , extracting them as events 1b to generate interleavings between the `Start` and `End` points.

From the extracted events, ER-π then starts working on generating interleavings. To prevent the interleaving number from combinatorial explosion (e.g., n events can result in $n!$ interleavings), ER-π features four pruning techniques: Event Grouping (clusters causally related events), Replica-Specific Merging (ignores irrelevant orders for replicas not under test), Event Independence (merges permutations of independent events), and Failed Operations (consolidates redundant interleavings of unsuccessful operations) [11]. These algorithms ensure that only necessary interleavings are explored. Each interleaving is timestamped to ensure proper execution order, then persisted in a Datalog-backed database for efficient storage and querying, 2 .

Then ER-π executes the generated interleavings one by one 3 . It checkpoints the replicas' states and resets them before executing each interleaving, ensuring that interleavings execute without interfering with each other. ER-π invokes interleaving events, enforcing the required event order via a distributed lock, provided by a distributed locking library [5]. The lock's mutex with a shared key ensures the required distributed order in each interleaving.

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. Provided as functions, the tests can be invoked after each interleaving. ER-π also provides the ability to specify custom test assertions. Developers can specify a custom test as a function that can be passed as a parameter to `ER-π.End()`.

Implementation: ER-π supports RDLs implemented in Go, Java, and JavaScript. To detect and replay the event interleavings without manual modification of the RDLs' source code, ER-π provides language-specific bindings. 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. Language-specific bindings are deliberately lightweight: fewer than 300 lines of Go (via AST rewriting [20]), 280 lines of JavaScript (via Monkey Patching [9]), and 415 lines of Java (via Dynamic Proxy [6]).

However, some functionalities of ER-π are language-agnostic, such as generating, pruning, and persisting interleavings, also controlling their event order. To serve as a reliable and efficient shared unit of functionality, these tasks are implemented in C++, known for its support for heterogeneity, equipped with a mature compilation and execution infrastructure that ensures portability across platforms [3]. Specifically, the language-agnostic functionalities are implemented in approximately 2K lines of C++ code. To store and query interleavings, ER-π leverages Datalog (Soufflé dialect) [13], enabling efficient pruning via logic-based queries over potentially factorial-sized interleaving sets.

2.2 Initial Results

To evaluate the effectiveness of ER-π, we applied it to four third-party replicated data systems implemented in Go, Java, and JavaScript: (1) Soundcloud's Roshi (Go) [2], (2) OrbitDB (JavaScript) [21], (3) ReplicaDB (Java) [16], and (4) Yorkie (Go) [7]. For each subject, we prototyped application logic that exercised their RDLs' functions, enabling systematic capture and replay of event interleavings.

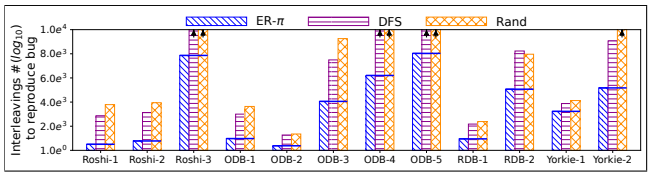
Reproducing Bugs: One key use case of ER-π is reproducing bugs that arise from subtle interleavings in deployed systems. When a bug is experienced during the execution of a replicated data system, it might be impossible for developers to report which of the possible interleavings was in effect when the bug manifested itself. To evaluate ER-π's effectiveness for this task, we reproduced 12 previously reported bugs across the four subjects, as listed in Table 1. Despite many of these bugs being fixed in current releases, ER-π successfully reproduced them by running earlier versions of the libraries. The bugs spanned multiple causes, including misconceptions, misuse, and RDL-specific issues, highlighting ER-π 's ability to capture a wide range of failure scenarios.

Reducing the Problem Space: To evaluate ER-π 's pruning capability, we compared it against exhaustive exploration strategies: depth-first search (DFS) and random exploration (Rand). For each bug listed in Table 1, we recorded the number of interleavings and time required for reproduction, with the results depicted in Figures 2a and 2b, respectively. Both DFS and Rand failed to reproduce certain bugs within 10K interleavings (e.g., Roshi-3, OrbitDB-4,

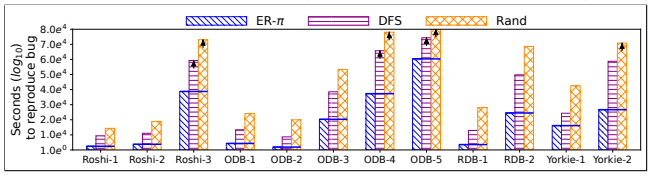
BugName	#Events	Status	Reason
Roshi-1	9	closed	misconception
Roshi-2	10	closed	RDL issue
Roshi-3	21	closed	misconception
OrbitDB-1	12	open	—
OrbitDB-2	8	open	—
OrbitDB-3	15	closed	misuse
OrbitDB-4	18	closed	misconception
OrbitDB-5	24	closed	misconception
ReplicaDB-1	10	closed	misuse
ReplicaDB-2	14	closed	misconception
Yorkie-1	17	open	—
Yorkie-2	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.

OrbitDB-5), whereas ER-π successfully reproduced all bugs. Moreover, ER-π consistently required fewer interleavings and significantly less time than the competitors. For example, ER-π reproduced bugs such as Roshi-3 and OrbitDB-5 in under 8K interleavings, while DFS and Rand either failed or required prohibitively long runtimes.



(a) Number of Interleavings Explored to Reproduce Bug



(b) Time Required to Reproduce Bug

Figure 2: Number of interleavings and time (both in log_{10} scale) required to reproduce bugs. ODB and RDB stand for OrbitDB and ReplicaDB, respectively. ↑ indicates that the bug is not reproduced after exploring $10K$ interleavings.

These initial results demonstrate that ER-π not only reproduces real-world bugs across diverse RDLs but also prunes the interleaving space into a manageable subset. This evidence positions ER-π as a practical integration testing solution for replicated data systems that bridges the gap between thoroughness and practicality.

3 ONGOING RESEARCH

In this section, we discuss my dissertation's ongoing research, concerned with prioritizing interleavings.

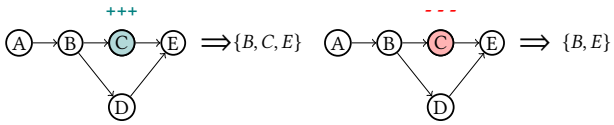
3.1 Motivation

While pruning reduces the interleaving search space, the remaining set can still be prohibitively large, often impossible to replay exhaustively within a reasonable time frame. My current research focuses on prioritizing interleavings so that those most likely to expose bugs introduced by recent code changes are replayed first. Consider a collaborative editing system where one replica introduces a new operation for formatting text. Without prioritization, ER-π might replay thousands of interleavings before reaching the ones involving this new operation. With prioritization, interleavings that include the new formatting events are weighted higher and replayed earlier, surfacing potential integration bugs faster. Another example is a key-value store in which developers have modified the conflict resolution for concurrent writes. Here, prioritization would highlight write–write races and resolution logic as directly impacted, steering replay toward interleavings with concurrent writes. By deprioritizing unrelated read–delete combinations, bugs in the new resolution mechanism can be exposed faster. These examples motivate why prioritization is important even with pruning: by accelerating bug discovery, it reduces system downtime, directing developers' attention to the interleavings that matter most.

3.2 Approach

To better explain the prioritization approach, we first introduce some terminology. Consider a replicated data system with application logic $\mathcal{A}$.

- **Component:** A *component* is any constituent unit of the application logic $\mathcal{A}$ (e.g., function, block).
- **Event:** An *event* e is an API invocation that occurs due to an interaction between the RDL and application code ($\mathcal{A}$). Let E represent the set of all possible events in $\mathcal{A}$.
- **Interleaving:** An *interleaving* is a specific sequence of events that occur during execution. Let $\mathcal{I}$ denote the set of all possible interleavings: $\mathcal{I} = \{i_1, i_2, \ldots, i_n\}$, where each i represents an ordered sequence of all events from E.
- **Sub-interleaving:** A *sub-interleaving* is a sequence that consists of a strict subset of events in E.
- **Impacted Components, Events, and Sub-interleavings:** A component is considered impacted if impact analysis [1] identifies that the component has introduced, modified, or removed an event in $\mathcal{A}$. Let C denote the set of impacted components in $\mathcal{A}$. For each impacted component $c \in C$, let $E_c \subseteq E$ represent the subset of impacted events associated with that component. We define an impacted sub-interleaving as a sequence of events that surround an impacted event.



(a) Positive Case **(b) Negative Case**

Figure 3: Sub-interleaving Construction from Call Graph

Consider the impacted event e_i. An impacted sub-interleaving is constructed from n events occurring before and after e_i in the

program's call graph [14], illustrated in Figure 3. Suppose impact analysis shows that an event has been added to the call graph. In that case, the sub-interleaving comprises the events immediately preceding and following the event, including the event itself. For instance, adding event `C` to the call graph would form the sub-interleaving `{B,C,E}` (Figure 3a). Conversely, if impact analysis reveals that an event has been removed from the call graph, the sub-interleaving comprises the events immediately before and after the event, excluding the event itself. For example, removing event `C` from the call graph would form the sub-interleaving `{B,E}` (Figure 3b).

- **Interleaving Weight:** The *interleaving weight* k denotes the number of impacted sub-interleavings present in it.

This approach aims to prioritize interleavings such that those most likely to trigger a bug are replayed first. Specifically, we seek an ordering $\mathcal{I}'$ of $\mathcal{I}$: $\mathcal{I}' = \langle i'_1, i'_2, \ldots, i'_n \rangle$ such that $\text{weight}(i'_1) \geq \text{weight}(i'_2) \geq \ldots \geq \text{weight}(i'_n)$. The approach follows three steps: (1) conduct impact analysis, (2) compute weights, and (3) prioritize and output $\mathcal{I}'$, depicted in Figure 4.

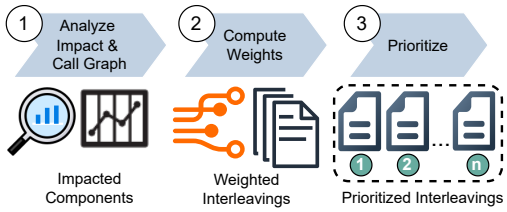


Figure 4: Prioritizing Interleavings

① **Impact analysis:** This step identifies the impacted components C and also the sets of events E_c impacted by each component in C. This task is accomplished using software change impact analysis techniques [15]. These analysis techniques aim to identify how changes to one part of software affect others. We are using a third-party impact analysis approach, such as Change Impact Analysis (CIA) [18], to identify impacted components and their respective sets of events.

② **Weight Computation:** This step calculates the weights of each possible interleaving in $\mathcal{I}$. The programming model that can be applied for this task remains an open question. Experimentation with ProbLog, a probabilistic logic programming language [8], seems promising, as this language's declarative model facilitates the expression of probabilistic reasoning required for this task.

③ **Prioritization:** The final step sorts each interleaving by weight in descending order to establish $\mathcal{I}'$. When the prioritization is complete, the interleavings will be replayed in this order.

Toward a unified solution: Both thrusts of my dissertation research—pruning and prioritization—will serve as building blocks for realizing the vision of providing a thorough and practical integration testing for replicated data systems. Once both are fully realized, then these thrusts should be usable individually à la carte, or in combination in any order, as required to satisfy the needs of particular testing scenarios. The concluding piece of my dissertation will determine how to apply these building blocks to achieve the required testing objectives.

4 CONCLUSION

My dissertation research addresses the challenges of exhaustive interleaving replay for replicated data systems by combining pruning and prioritization. The completed research, ER-π, has demonstrated practical benefits in uncovering integration bugs. The ongoing prioritization work aims at making interleaving-based testing both thorough and efficient. Once fully realized, these contributions would advance integration testing support for replicated data systems, thus harmoniously fusing thoroughness and practicality.

ACKNOWLEDGMENT

I sincerely thank my advisor, Prof. Dr. Eli Tilevich, for his continuing support and guidance.

REFERENCES

[1] Robert S Arnold. 1996. *Software change impact analysis.* IEEE Computer Society Press.
[2] Peter Bourgon and Nick Stenning. 2014. Roshi. https://github.com/soundcloud/roshi.
[3] 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.
[4] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated data types: specification, verification, optimality. *ACM Sigplan Notices* 49, 1 (2014), 271–284.
[5] Redis Documentation. 2023. Distributed Locks with Redis. https://redis.io/docs/latest/develop/use/patterns/distributed-locks/.
[6] 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.
[7] Youngteac Hong and Dongcheol Choe. 2020. Yorkie. https://github.com/yorkie-team/yorkie.
[8] Angelika Kimmig, Bart Demoen, Luc De Raedt, Vitor Santos Costa, and Ricardo Rocha. 2011. On the implementation of the probabilistic logic programming language ProbLog. *Theory and Practice of Logic Programming* 11 (2011), 235–262.
[9] 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.
[10] Provakar Mondal and Eli Tilevich. 2023. Undoing CRDT Operations Automatically. In *2023 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 246–251.
[11] Provakar Mondal and Eli Tilevich. 2025. ER-π: Exhaustive Interleaving Replay for Testing Replicated Data Library Integration. In *Proceedings of the 26th International Middleware Conference*.
[12] Zvonimir Rakamarić. 2010. STORM: static unit checking of concurrent programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. 519–520.
[13] Yann Ramusat, Silviu Maniu, and Pierre Senellart. 2022. Efficient provenance-aware querying of graph databases with datalog. In *Proceedings of the 5th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. 1–9.
[14] Barbara G Ryder. 2006. Constructing the call graph of a program. *IEEE Transactions on Software Engineering* 3 (2006), 216–226.
[15] Soobia Saeed, NZ Jhanjhi, Mehmood Naqvi, and Mamoona Humayun. 2019. Analysis of software development methodologies. *International Journal of Computing and Digital Systems* 8, 5 (2019), 446–460.
[16] Oscar Salvador and Francesco Zanti. 2018. ReplicaDB. https://github.com/osalvador/ReplicaDB.
[17] 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.
[18] Xiaobing Sun, Bixin Li, Chuanqi Tao, Wanzhi Wen, and Sai Zhang. 2010. Change impact analysis based on a taxonomy of change types. In *2010 IEEE 34th Annual computer software and applications conference*. IEEE, 373–382.
[19] Henrik Thane. 2000. *Monitoring, testing and debugging of distributed real-time systems.* Ph. D. Dissertation. Maskinkonstruktion.
[20] Adam Welc. 2021. Automated Code Transformation for Context Propagation in Go. In *Proceedings of the 29th ACM Joint Meeting on European Software Eng. Conference and Symposium on the Foundations of Software Eng.* 1242–1252.
[21] Friedel Ziegelmayer and Hayden Young. 2018. OrbitDB. https://github.com/orbitdb/orbitdb.