Sal: Multi-modal Verification of Replicated Data Types

Pranav Ramesh
Indian Institute of Technology, Madras
Chennai, India
cs22b015@smail.iitm.ac.in

Vimala Soundarapandian
Indian Institute of Technology, Madras
Chennai, India
cs19d750@smail.iitm.ac.in

KC Sivaramakrishnan
Indian Institute of Technology, Madras
Chennai, India
kcsrk@cse.iitm.ac.in

Abstract

Designing correct replicated data types (RDTs) is challenging because replicas evolve independently and must be merged while preserving application intent. A promising approach is correct-by-construction development in a proof-oriented programming language such as F⋆, Dafny and Lean, where desired correctness guarantees are specified and checked as the RDTs are implemented. Recent work Neem [19] proposes the use of replication-aware linearizability (RA linearizability) [20] as the correctness condition for state-based CRDTs and mergeable replicated data types (MRDTs), with automation in the SMT-aided, proof-oriented programming language F⋆. However, SMT-centric workflows can be opaque when automation fails to discharge a verification condition (VC), and they enlarge the trusted computing base (TCB).

We present Sal, a multi-modal workflow to design and verify state-based CRDTs and MRDTs in Lean. Sal combines (i) kernel-checkable automation with proof reconstruction, (ii) SMT-aided automation when needed, and (iii) interactive theorem proving for remaining proof obligations. When automated verification fails, we leverage Lean's property-based testing to automatically generate and visualize counterexamples, helping developers debug incorrect specifications or implementations. We report on our experience verifying a suite of 13 CRDTs and MRDTs with Sal: 69% of verification conditions are discharged by kernel-verified automation without SMT, and counterexamples automatically expose subtle bugs such as the well-known enable-wins flag anomaly. The codebase for Sal is open-sourced, and is available at https://github.com/fplaunchpad/sal.

CCS Concepts: • **Software and its engineering → Software verification and validation**; • **Computing methodologies → Distributed algorithms**.

Keywords: CRDT, Verification, Lean, Multi-Modal Proofs, Counterexample Generation

1 Introduction

Local-first collaboration tools allow users to continue working while offline, synchronizing in the background when connectivity returns [10]. This programming model requires replicated data types whose states can evolve independently at each replica and later be merged without violating application intent. Conflict-Free Replicated Data Types (CRDTs) [18] and Mergeable Replicated Data Types (MRDTs) [8] are widely used to implement such replicated state. However, designing correct RDTs is subtle. Even well-known designs such as Replicated Growable Arrays (RGA) [9] have had serious anomalies discovered after publication[1].

Replication-aware (RA) linearizability [20] provides a principled correctness condition for CRDTs, relating replica executions to a sequential explanation of updates. Prior work, Neem [19], showed that RA linearizability can be extended to MRDTs and also be reduced to a set of 24 verification conditions (VCs) amenable to automation. Neem implements this approach in F⋆, an SMT-aided, proof-oriented programming language. Developers implement RDTs in F⋆ and leverage SMT automation to discharge the RA-linearizability VCs automatically. In practice, however, SMT-centric verification workflows can be difficult to iterate on. When automation fails, developers often get little actionable information beyond an unproved VC, and debugging incorrect implementations remains manual and time-consuming.

This debugging experience contrasts sharply with everyday software development, where programmers iterate quickly by writing code, running tests, inspecting failing traces, and refining implementations. Understanding why a particular VC fails requires a laborious "proof-debugging" workflow: developers add intermediate assertions (`assert`) or assumptions (`admit`) and progressively push them deeper to localize the failing reasoning step. When the root cause is an unexpected interaction between definitions and the SMT encoding, additional tuning (e.g., controlling unfolding or solver limits) may be needed. Moreover, SMT-aided proofs can be brittle: small changes may cause previously discharged VCs to fail. Relying on an external SMT solver also increases the trusted computing base.

We present Sal, a verification workflow for RA-linearizability in Lean that addresses these issues by making failures actionable. At the core of Sal is a staged Lean tactic that prioritizes proof reconstruction (e.g., simplifying goals and invoking `grind` [12]) and falls back to SMT-aided automation (`lean-blaster` [7]) only when necessary. When a VC is false, Sal uses property-based testing (Plausible [13]) to automatically synthesize small counterexamples and a ProofWidgets [1]-based visualizer to render the corresponding execution trace, enabling a test-like debugging loop for RDT verification. In our evaluation over a suite of CRDTs and

[1] https://martin.kleppmann.com/2019/03/25/papoc-interleaving-anomalies.html#errata

MRDTs, Sal discharges most VCs without SMT and automatically rediscovers subtle bugs such as the well-known enable-wins flag anomaly (Table 2).

This paper makes the following contributions:

1. A Lean formalization of RA-linearizability VCs for a suite of state-based CRDTs and MRDTs.
2. A counterexample-generation and visualization workflow for failing VCs, based on property-based testing in Lean.
3. Sal, a custom tactic that attempts proof reconstruction-based automation first and falls back to SMT-aided automation, with interactive proving as a last resort.
4. An evaluation across a suite of CRDTs and MRDTs, including cases where counterexamples expose subtle bugs (Table 2).

2 Background

We assume familiarity with replicated data types and focus on the specific models and specifications used in this paper. State-based CRDTs reconcile replicas via a deterministic two-way merge $\mu(v_1, v_2)$; a common sufficient condition for convergence is that the merge function μ is the join of a semilattice.

Mergeable Replicated Data Types (MRDTs) [8] avoid embedding causal metadata in the state of the RDT. Instead, they assume a versioned storage model (e.g., Git-like histories) that can provide the lowest common ancestor (LCA) for a three-way merge. This interface often yields compact implementations: for example, a counter MRDT can be $O(1)$, whereas state-based counter CRDTs require $\Omega(n)$ space in the number of replicas [2].

Formally, an MRDT implementation for a data type τ [19] is a tuple $\mathcal{D}_\tau = \langle \Sigma, \sigma_0, \mathsf{do}, \mathsf{merge}, \mathsf{rc} \rangle$, where:

- Σ is the set of states, $\sigma_0 \in \Sigma$ is the initial state.
- $\mathsf{do} : \Sigma \times \mathcal{T} \times \mathcal{R} \times O_\tau \to \Sigma$ implements update operations parameterized by timestamp in $\mathcal{T}$, replica id in $\mathcal{R}$ and operations in O_τ.
- $\mathsf{merge} : \Sigma \times \Sigma \times \Sigma \to \Sigma$ is a three-way merge function.
- $\mathsf{rc} \subseteq O_\tau \times O_\tau$ is the conflict resolution policy to be followed for concurrent *conflicting* update operations. The relation is interpreted as a partial order where $(o_1, o_2) \in \mathsf{rc}$ means that o_1 is ordered before o_2.

For example, the increment-only counter MRDT is defined as follows:

- $\Sigma = \mathbb{N}$ with $\sigma_0 = 0$
- $O = \{\mathsf{inc}\}$
- $\mathsf{do}(\sigma, \_, \_, \mathsf{inc}) = \sigma + 1$
- $\mathsf{merge}(\sigma_{lca}, \sigma_1, \sigma_2) = \sigma_{lca} + (\sigma_{lca} - \sigma_1) + (\sigma_{lca} - \sigma_2)$ – the merged state is the sum of the states of the LCA and the difference between the LCA and the two versions.
- $\mathsf{rc} = \emptyset$ – no conflicting operations.

An observed-removed set (OR-set) MRDT [8] is defined as follows:

- $\Sigma = \mathcal{P}(\mathcal{T} \times E)$ where E is the set of elements with $\sigma_0 = \emptyset$
- $O = \{\mathsf{add}_e, \mathsf{rem}_e | e \in E\}$
- $\mathsf{do}(\sigma, t, \_, \mathsf{add}_e) = \sigma \cup \{(e, t)\}$
- $\mathsf{do}(\sigma, \_, \_, \mathsf{rem}_e) = \sigma \setminus \{(e, i) | (e, i) \in \sigma\}$
- $\mathsf{merge}(\sigma_{lca}, \sigma_1, \sigma_2) = (\sigma_{lca} \cap \sigma_1 \cap \sigma_2) \cup (\sigma_1 \setminus \sigma_{lca}) \cup (\sigma_2 \setminus \sigma_{lca})$ – the merged state contains elements common to all three versions (unchanged elements remain; deleted elements are not included), as well as elements added in either version since the LCA.
- $\mathsf{rc} = \{(\mathsf{rem}_e, \mathsf{add}_e) | e \in E\}$ – remove is ordered before add for the same element, and hence, adds win over concurrent removes.

We verify RDT correctness using replication-aware (RA) linearizability [20], which makes merge semantics explicit while retaining a linearizability-style reading. Neem [19] reduces RA linearizability for MRDTs and state-based CRDTs to a finite set of verification conditions (VCs) over do, merge, and rc. Sal focuses on discharging these VCs effectively and on producing actionable feedback when automation fails.

Lean 4 [16] is a theorem prover and functional programming language with a small trusted kernel; proofs can be checked as proof terms by the kernel. Lean's metaprogramming support enables custom tactics and domain-specific automation, which is useful for VC-heavy developments like ours.

Lean offers several automation techniques with different tradeoffs. We rely on the following:

- `dsimp` and `aesop`: simplification and proof search for routine goals.
- `grind` [12]: SMT-style automation with proof reconstruction, producing kernel-checkable proof terms.
- `lean-blaster` [7]: an SMT backend (Z3) that is effective on many VCs (notably those with lambdas), but without proof reconstruction; it therefore enlarges the trusted computing base.

Sal stages automation accordingly: we first attempt reconstructible automation and only fall back to SMT when needed.

3 Sal framework

In this section, we describe the Sal multi-modal verification framework for RA-linearizability in Lean. We start with the challenge of designing data structures conducive to automated verification in Lean.

3.1 Data structures for automated verification

Many RDTs rely fundamentally on sets and maps. For example, the OR-set MRDT maintains a set of timestamped elements; map-based MRDTs and CRDTs require reasoning about extensional equality of mappings. However, Lean's

standard library provides data structures optimized for different purposes: the mathematical Set type uses propositions ($\alpha \to$ Prop) suitable for manual proofs, while computational maps like RBMap and HashMap emphasize efficient iteration but complicate extensional reasoning.

For automated verification, we use decidable representations that tools like `grind` can reason about effectively. We therefore implement custom set and map interfaces inspired by F⋆'s verification-oriented designs. Our sets use boolean-valued membership functions rather than propositions, since they typically contain datatypes where equality is decidable.

```
abbrev set (a:Type) [DecidableEq a] := a → Bool
```

This representation is decidable by construction: since membership returns a boolean, automation can directly compute and compare set operations without requiring proof objects to reason about decidability. We require DecidableEq for element types, ensuring all operations remain computable. This is a natural condition which is satisfied across the RDTs we verify with this framework.

Similarly, our map interface makes the domain explicit to enable extensional reasoning:

```
structure map (key:Type) [DecidableEq key]
    (value:Type) where
  mappings: key → value
  domain: set key
```

Two maps are equal when their domains and mappings agree. Since the mapping is restricted to the domain key which supports decidable equality, the map supports extensional equality.

Critically, we annotate all definitions and lemmas with @[simp, grind] attributes and provide grind_pattern hints. This builds a domain-specific rewrite database: when `grind` encounters goals involving set membership, union, or map selection, it automatically applies the appropriate lemmas without manual guidance. These annotations trade the generality of Lean's standard library for automation-friendliness, enabling `grind` to discharge most VCs involving sets and maps without SMT assistance (Table 2). This is analogous to SMT patterns in F⋆.

3.2 Counterexample generation in Lean

When a VC fails during verification, understanding why it failed is critical for debugging. In SMT-centric workflows like F⋆, a failed VC provides little actionable feedback: developers must manually construct execution traces, add intermediate assertions, and progressively narrow down the source of the failure. This process is labor-intensive and requires significant expertise. In contrast, Sal leverages Lean's property-based testing framework, Plausible [13], to automatically generate concrete counterexamples when VCs fail, transforming opaque proof failures into tangible test cases that developers can inspect and debug.

```
abbrev concrete_st := Int × Bool
inductive app_op_t : Type where
| Enable
| Disable
abbrev op_t:= ℕ × ℕ × app_op_t
  /-timestamp, rid, operation-/
def do_ (s:concrete_st) (o: op_t) : concrete_st
:= match o with
| (_, (_, .Enable)) => (Prod.fst s + 1, true)
| (_, (_, .Disable)) => (Prod.fst s, false)
def merge_flag (l a b: concrete_st) :=
  if Prod.snd a && Prod.snd b then true
  else if not (Prod.snd a) && not (Prod.snd b) then
    false
  else if Prod.snd a then Prod.fst a > Prod.fst l
  else Prod.fst b > Prod.fst l
def merge (l a b: concrete_st) : concrete_st
:= (Prod.fst a + Prod.fst b - Prod.fst l ,
    merge_flag l a b)
```

Figure 1. Buggy enable-wins flag MRDT implementation

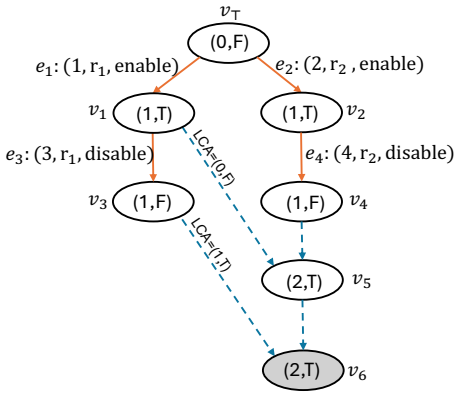

Figure 2. An enable-wins flag execution: both replicas see a disable at the end, yet merging produces $(2, \text{true})$ at v_6, incorrectly reporting the flag as enabled.

We demonstrate this approach using the enable-wins flag MRDT, a shared boolean flag that represents a disabled or enabled state. The desired specification is that in the case of concurrent enable and disable operations, the enable wins. This is expressed as disable $\xrightarrow{\text{rc}}$ enable, meaning that when we read a replica's state, the flag should be true if there exists an enable operation that is not causally preceded by a disable.

Consider the enable-wins flag implementation in Figure 1, which tracks the number of concurrent enables using a counter and uses this counter to determine the flag's state after merging. This implementation contains a subtle bug. The bug manifests in executions where enable operations are followed by disables on each replica, yet the merged

state incorrectly reports the flag as enabled. Figure 2 shows such an execution: when merging v_3 and v_5 (with LCA v_1), the counter value of v_5 exceeds v_1, causing merge_flag to compute the new flag to be true in v_6. However, all enable events in this execution are subsequently disabled on their respective replicas, violating the enable-wins specification! When verifying the RA-linearizability VCs, one of the VCs fails. However, finding such counterexamples manually is challenging, particularly for more complex RDTs with larger state spaces.

To automate counterexample discovery, we leverage Plausible [13], a property-based testing tool for Lean inspired by QuickCheck [3]. Plausible requires that VCs be expressed as decidable propositions – that is, properties that can be algorithmically evaluated to true or false. Once decidability is established, Plausible generates random test inputs and checks whether they satisfy the VC. When a violation is found, Plausible reports the failing input, such as:

```
postcondition violated for input
(disable, (enable, (enable, (disable, (0, false)))))
```

While this output identifies a counterexample, diagnosing the cause of the failure is difficult. To address this, we implement an execution trace visualizer using Lean's ProofWidgets framework [1]. The visualizer instruments the do and merge operations to record intermediate states and operations, producing a step-by-step execution trace.

Sal, like Neem [19], includes VCs that check whether concurrent executions can be RA-linearized to the same final state. This is achieved using *bottom-up linearization*, where we peel off events in a bottom-up manner to construct a linearization order. In the failed VC, we attempt to prove:

$$\mu\big(\overbrace{e_1(0,false)}^{v_1},\ \overbrace{e_3(e_1(0,false))}^{v_3},\ \overbrace{e_1(e_4(e_2(0,false)))}^{v_5}\big) = e_3\big(\mu(\underbrace{e_1(0,false)}_{v_1},\ \underbrace{e_1(0,false)}_{v_1},\ \underbrace{e_1(e_4(e_2(0,false)))}_{v_5})\big)$$

where μ is the merge operation, $e_1 \ldots e_4$ are events, and each tuple (Int × Bool) constitutes a state. The state at v_6 in Figure 2 is obtained by merging v_3 and v_5, with v_1 as LCA. This corresponds to the LHS of the VC. This state should match the state obtained by first merging v_1 and v_5, with v_1 as LCA, and then applying e_3, through bottom-up linearization.

Figure 3 illustrates the visualization for the enable-wins flag that violates this VC. The states and operations are shown in blue and yellow boxes, respectively, and correspond to concrete_st and and op_t, respectively, from Figure 1. The LCA trace and state is shown at the top. In both cases, the LCA state is v_1. The left panel (a) shows the execution trace for the LHS, which evaluates to (2, true), and the right panel (b) shows the RHS, which evaluates to (2, false). By comparing these traces, developers can quickly identify

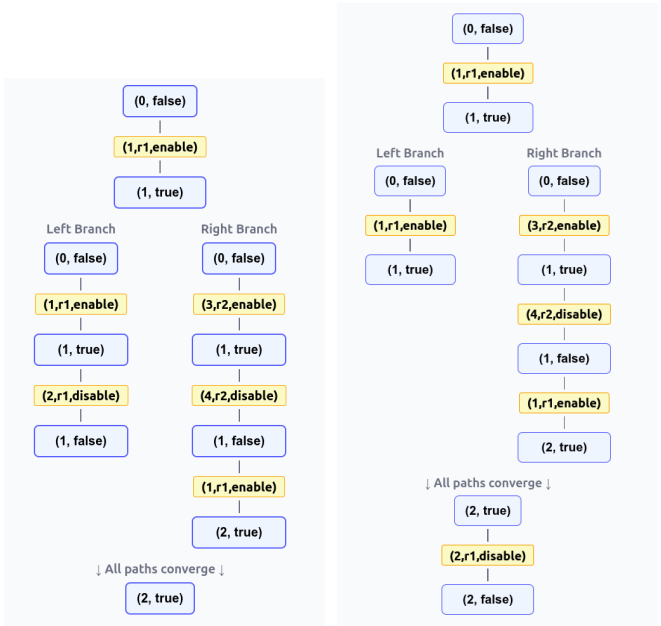

Figure 3. Visualization for the failed VC for the buggy enable-wins flag in Figure 1.

that the bug arises from the counter-based merge logic failing to account for subsequent disables. This visualization transforms the abstract VC failure into a concrete debugging scenario, analogous to examining a failing unit test trace in conventional software development.

The combination of automatic counterexample generation and interactive visualization significantly accelerates the debugging workflow. Instead of manually inspecting failed VCs and constructing hypothetical execution traces, developers receive concrete, visualizable counterexamples automatically. This approach also complements the multi-modal proof strategy: when `grind` or `lean-blaster` fail to discharge a VC, Plausible can quickly determine whether the failure stems from an incorrect implementation (producing a counterexample) or requires manual interactive proof (when no counterexample exists).

3.3 Visualizing functional sets

While the enable-wins flag uses simple concrete types (integers and booleans), many RDTs are set-based. Our custom sets (Section 3.1), designed as functional predicates $a \to \text{Bool}$ for verification, pose a visualization challenge: they are abstract, infinite by nature, and do not support iteration or enumeration. Yet both counterexample debugging and general trace inspection require displaying actual set contents as concrete element lists.

To bridge this gap, we implement a universe tracking mechanism. During execution, we maintain a finite HashSet

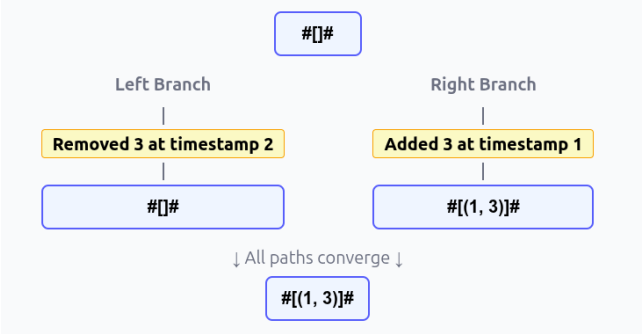

Figure 4. OR-set execution visualized using ProofWidgets

of all elements that have been added or removed. When visualizing the set state, we only check membership for elements in this finite universe, which suffices to characterize the set's observable behavior. The implementation augments our abstract sets with a concrete universe:

```
structure set_with_universe (α: Type) [ToString α]
    [DecidableEq α] [Hashable α] where
  _set : set α
  _universe : HashSet α
```

Figure 4 demonstrates this approach on an OR-set execution with concurrent add and remove operations on element 3. The visualization shows concrete set states using the notation #[(1, 3)]#, where the tuple contains the timestamp and element. The left branch removes element 3 (resulting in the empty set #[]#), while the right branch adds element 3 (resulting in #[(1, 3)]#). Both branches converge to #[(1, 3)]#, confirming that adds win over concurrent removes as specified. Operations are displayed in yellow boxes, and users provide operation labels via format strings.

This mechanism works uniformly for both correct executions and counterexamples, enabling developers to inspect set-based RDT behavior regardless of whether they are debugging a failed VC or validating a correct implementation.

3.4 Multi-modal proofs using the Sal tactic

The Sal tactic orchestrates a staged proof strategy that adapts to VC complexity. As described in Section 2, Lean provides multiple automation techniques with different trade-offs between power, trust, and performance. The Sal tactic attempts these approaches sequentially, prioritizing proof reconstruction before falling back to more powerful but less trustworthy methods.

The tactic proceeds in the following stages:

1. `dsimp + grind (DG)` – This combination applies simplification followed by `grind`'s SMT-style automation with proof reconstruction, producing kernel-checkable proof terms while maintaining the smallest TCB. We exclude `aesop` [14] due to prohibitively high verification times on RDT VCs.
2. `lean-blaster (LB)` – When DG fails, `lean-blaster` encodes the goal to Z3. While more powerful for VCs with higher-order functions and lambdas, this sacrifices proof reconstruction and enlarges the TCB. We select it over Lean-SMT [15] and Lean-Auto [17] for its superior support for higher-order functions.

If both automated stages fail and no counterexample was generated (Section 3.2), the VC requires interactive proving. Developers must then manually construct proofs using Lean's tactic language. To prevent runaway automation, the Sal tactic incorporates a heartbeat-based timeout mechanism that bounds the execution time of each stage. When a timeout expires, control returns to the user, allowing them to either increase the limit or proceed directly to interactive proving.

4 Evaluation

Table 1 compares F* and Lean across several dimensions relevant to RDT verification. Lean's rich tactic system enables the multi-modal workflow central to Sal, while built-in counterexample generation (Plausible) and proof reconstruction support actionable debugging and reduced TCB. However, Lean's standard library data structures are designed for mathematical reasoning rather than automation, necessitating the custom sets and maps described in Section 3.1.

Table 2 shows the results of verifying 13 RDTs (312 VCs in mtotal) using Sal. Across all benchmarks, **215 VCs (68.9%) are discharged by dsimp+grind**, a lightweight tactic with proof reconstruction verified by Lean's kernel. An additional 87 VCs (27.9%) require lean-blaster (SMT-based), and only 9 VCs (3%) fall back to interactive proving. This demonstrates that the majority of RA-linearizability VCs can be verified without enlarging the TCB through SMT solvers.

We observe notable differences between CRDTs and MRDTs. MRDTs generally require less SMT than CRDTs due to their simpler three-way merge; for example, PN-counter MRDT needs no lean-blaster while PN-counter CRDT requires it for 2 VCs plus 6 interactive proofs. The 9 interactive proofs are concentrated in map-based RDTs, reflecting that Lean's map automation is less mature than its set automation.

The enable-wins flag MRDT demonstrates the value of counterexample generation. This implementation contains a known bug from prior work [19]; when the corresponding VC fails, Plausible automatically generates a concrete counterexample, which our ProofWidgets-based visualizer renders for inspection (Figure 3). This workflow transforms opaque VC failures into actionable debugging scenarios.

5 Related work

Multi-modal verification. Loom [5] is a recent work on multi-modal verification for data structures in Lean, using Dijkstra Monads to model effectful operations on mutable arrays and counters. Sal adapts this multi-modal approach

Table 1. Comparing F* and Lean for verifying RA Linearizability

Criterion	F*	Lean
Automation	Direct SMT solving via Z3. Automation often works out of the box at scale.	Multiple tactic-based tools (`aesop`, `grind`, `lean-blaster`), less effective on large proof goals.
Multi-Modal Proofs	Limited tactic system; most proofs require SMT solver.	Rich tactic system allows combining automated and interactive proving.
Counterexamples	No counterexample generation; reports only success or failure.	`Plausible` generates counterexamples for decidable properties.
Data Structures	Rich set and map libraries designed for verification.	Mathematical data structures; custom sets and maps needed for automation (Section 3.1).
Trustworthiness	No proof reconstruction; relies on trusting Z3.	Proof reconstruction for most tools; kernel verifies proofs. Uncertified proofs marked `sorry`.

Table 2. Number of VCs discharged by the Sal tactic (total VCs = 24 per RDT). DG refers to dsimp + grind, and LB refers to lean − blaster. #Contains known bug; counterexample automatically generated (Section 3.2).

RDT	DG	LB	Fallback to ITP
Increment-only counter MRDT	24	0	0
PN-counter MRDT	24	0	0
OR-set MRDT	3	21	0
Enable-wins flag MRDT #	9	14	0
Efficient OR-set MRDT	2	22	0
Grows-only set MRDT	24	0	0
Grows-only map MRDT	22	0	2
Replicated growable array MRDT	15	9	0
Multi-valued register MRDT	24	0	0
Increment-only counter CRDT	24	0	0
PN-counter CRDT	16	2	6
Multi-valued register CRDT	24	0	0
OR-set CRDT	4	19	1

to RDT verification in Lean, but leverages conventional tactics rather than monadic effects since RDTs are monotonic and non-mutable. Unlike Loom's focus on sequential data structures, Sal targets the unique challenges of replicated systems with merge operations and causal reasoning.

CRDT verification. Several approaches verify CRDT correctness. Isabelle/HOL has been used to verify strong eventual consistency [6], but requires substantial manual proof effort. Neem [19] automates RA-linearizability verification in F* using SMT solvers, achieving high automation but at the cost of an enlarged TCB and opaque failures. Verifx [4] provides a specialized language for RDTs with built-in verification, trading generality for domain-specific automation. Sal distinguishes itself through kernel-verified automation (69% of VCs discharged by proof reconstruction), automated counterexample generation when verification fails, and interactive visualization of execution traces–capabilities absent in prior CRDT verification tools.

Counterexample generation. Property-based testing tools like QuickCheck [3], QuickChick [11] (Coq), and Plausible (Lean) generate test inputs for executable properties. Sal leverages Plausible for decidable VCs, but extends it with domain-specific visualization for RDT execution traces, including handling of functional set representations through universe tracking (Section 3.3). This bridges property testing and formal verification, transforming failed VCs into debuggable counterexamples.

6 Conclusion

We present Sal, a multi-modal verification framework for replicated data types in Lean that addresses key limitations of SMT-centric verification workflows. By staging automation–prioritizing kernel-verified proof reconstruction via `grind`, falling back to SMT-based `lean-blaster`, and supporting interactive proving for complex obligations–Sal achieves 69% kernel-verified automation across 13 CRDTs and MRDTs while minimizing the trusted computing base. When automation fails, automated counterexample generation via Plausible and interactive visualization through ProofWidgets transform opaque proof failures into actionable debugging scenarios, as demonstrated by rediscovering the enable-wins flag bug.

Our experience reveals patterns in RDT verification: MRDTs generally require less SMT than CRDTs due to simpler three-way merge, and map-based reasoning remains more challenging than set-based reasoning for current Lean automation. Future work includes extending our custom data structure library to improve map automation, developing optimizations for the staged tactic based on VC patterns, and evaluating Sal on larger-scale RDT developments.

References

[1] Edward W. Ayers, Mateja Jamnik, and W. T. Gowers. 2021. A Graphical User Interface Framework for Formal Verification. In *12th International Conference on Interactive Theorem Proving (ITP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 193)*, Liron Cohen and Cezary Kaliszyk (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 4:1–4:16. doi:10.4230/LIPIcs.ITP.2021.4

[2] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated data types: specification, verification, optimality. *SIGPLAN Not.* 49, 1 (Jan. 2014), 271–284. doi:10.1145/2578855.2535848

[3] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. *SIGPLAN Not.* 35, 9 (Sept. 2000), 268–279. doi:10.1145/357766.351266

[4] Kevin De Porre, Carla Ferreira, and Elisa Gonzalez Boix. 2023. VeriFx: Correct Replicated Data Types for the Masses. In *37th European Conference on Object-Oriented Programming (ECOOP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 9:1–9:45. doi:10.4230/LIPIcs.ECOOP.2023.9

[5] Vladimir Gladshtein, George Pîrlea, Qiyuan Zhao, Vitaly Kurin, and Ilya Sergey. 2026. Foundational Multi-Modal Program Verifiers. *Proc. ACM Program. Lang.* 10, POPL, Article 77 (Jan. 2026), 32 pages. doi:10.1145/3776719

[6] Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. 2017. Verifying strong eventual consistency in distributed systems. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 109 (Oct. 2017), 28 pages. doi:10.1145/3133933

[7] Input Output Global. 2024. Lean-blaster: An SMT-based proof automation tactic for Lean 4. GitHub repository. https://github.com/input-output-hk/Lean-blaster

[8] Gowtham Kaki, Swarn Priya, KC Sivaramakrishnan, and Suresh Jagannathan. 2019. Mergeable replicated data types. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 154 (Oct. 2019), 29 pages. doi:10.1145/3360580

[9] Martin Kleppmann, Victor B. F. Gomes, Dominic P. Mulligan, and Alastair R. Beresford. 2019. Interleaving anomalies in collaborative text editors. In *Proceedings of the 6th Workshop on Principles and Practice of Consistency for Distributed Data* (Dresden, Germany) *(PaPoC '19)*. Association for Computing Machinery, New York, NY, USA, Article 6, 7 pages. doi:10.1145/3301419.3323972

[10] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. 2019. Local-first software: you own your data, in spite of the cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Athens, Greece) *(Onward! 2019)*. Association for Computing Machinery, New York, NY, USA, 154–178. doi:10.1145/3359591.3359737

[11] Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. 2019. Coverage guided, property based testing. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 181 (Oct. 2019), 29 pages. doi:10.1145/3360607

[12] Lean Community. 2024. The grind tactic. Lean 4 Reference Manual. https://lean-lang.org/doc/reference/latest/The--grind--tactic/

[13] Lean Community. 2024. Plausible: A counterexample generator for Lean 4. Lean Reservoir. https://reservoir.lean-lang.org/@leanprover-community/plausible

[14] Jannis Limperg and Asta Halkjær From. 2023. Aesop: White-Box Best-First Proof Search for Lean. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Boston, MA, USA) *(CPP 2023)*. Association for Computing Machinery, New York, NY, USA, 253–266. doi:10.1145/3573105.3575671

[15] Abdalrhman Mohamed, Tomaz Mascarenhas, Harun Khan, Haniel Barbosa, Andrew Reynolds, Yicheng Qian, Cesare Tinelli, and Clark Barrett. 2025. lean-smt: An SMT Tactic for Discharging Proof Goals in Lean. In *Computer Aided Verification: 37th International Conference, CAV 2025, Zagreb, Croatia, July 23-25, 2025, Proceedings, Part III* (Zagreb, Croatia). Springer-Verlag, Berlin, Heidelberg, 197–212. doi:10.1007/978-3-031-98682-6_11

[16] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 625–635. doi:10.1007/978-3-030-79876-5_37

[17] Yicheng Qian, Joshua Clune, Clark Barrett, and Jeremy Avigad. 2025. Lean-Auto: An Interface Between Lean 4 and Automated Theorem Provers. In *Computer Aided Verification*, Ruzica Piskac and Zvonimir Rakamarić (Eds.). Springer Nature Switzerland, Cham, 175–196.

[18] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems*, Xavier Défago, Franck Petit, and Vincent Villain (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 386–400.

[19] Vimala Soundarapandian, Kartik Nagar, Aseem Rastogi, and KC Sivaramakrishnan. 2025. Automatically Verifying Replication-Aware Linearizability. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 111 (April 2025), 27 pages. doi:10.1145/3720452

[20] Chao Wang, Constantin Enea, Suha Orhun Mutluergil, and Gustavo Petri. 2019. Replication-aware linearizability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 980–993. doi:10.1145/3314221.3314617