Katara: Synthesizing CRDTs with Verified Lifting

SHADAJ LADDAD, University of California, Berkeley, USA
CONOR POWER, University of California, Berkeley, USA
MAE MILANO*, University of California, Berkeley, USA
ALVIN CHEUNG, University of California, Berkeley, USA
JOSEPH M. HELLERSTEIN*, University of California, Berkeley, USA

Conflict-free replicated data types (CRDTs) are a promising tool for designing scalable, coordination-free distributed systems. However, constructing correct CRDTs is difficult, posing a challenge for even seasoned developers. As a result, CRDT development is still largely the domain of academics, with new designs often awaiting peer review and a manual proof of correctness. In this paper, we present Katara, a program synthesis-based system that takes sequential data type implementations and automatically synthesizes verified CRDT designs from them. Key to this process is a new formal definition of CRDT correctness that combines a reference sequential type with a lightweight ordering constraint that resolves conflicts between non-commutative operations. Our process follows the tradition of work in verified lifting, including an encoding of correctness into SMT logic using synthesized inductive invariants and hand-crafted grammars for the CRDT state and runtime. Katara is able to automatically synthesize CRDTs for a wide variety of scenarios, from reproducing classic CRDTs to synthesizing novel designs based on specifications in existing literature. Crucially, our synthesized CRDTs are fully, automatically verified, eliminating entire classes of common errors and reducing the process of producing a new CRDT from a painstaking paper proof of correctness to a lightweight specification.

CCS Concepts: • **Software and its engineering → Automatic programming**; • **Computing methodologies → Distributed computing methodologies**.

Additional Key Words and Phrases: program synthesis, distributed systems, verification, replication

ACM Reference Format:
Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, and Joseph M. Hellerstein. 2022. Katara: Synthesizing CRDTs with Verified Lifting. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 173 (October 2022), 29 pages. https://doi.org/10.1145/3563336

1 INTRODUCTION

In today's interconnected world, there is an ever-growing need to write correct, scalable distributed programs that can serve users at any location with low latency. Many such applications rely on *distributed state*, which in turn is often *replicated* at multiple locations. Replication addresses many common concerns in distributed systems: it can lower latency by keeping a copy of data close to each client, improve availability by increasing the odds that some replica is on a reachable machine,

*also at Sutter Hill Ventures

Authors' addresses: Shadaj Laddad, University of California, Berkeley, USA, shadaj@cs.berkeley.edu; Conor Power, University of California, Berkeley, USA, conorpower@cs.berkeley.edu; Mae Milano, University of California, Berkeley, USA, mpmilano@cs.berkeley.edu; Alvin Cheung, University of California, Berkeley, USA, akcheung@cs.berkeley.edu; Joseph M. Hellerstein, University of California, Berkeley, USA, hellerstein@cs.berkeley.edu.

2475-1421/2022/10-ART173
https://doi.org/10.1145/3563336

and enhance the scalability of request handling by allowing the overall load of requests to be partitioned across replicas.

However, programmers are trained to write sequential programs and often struggle to write correct distributed programs that make concurrent updates to replicated state. As programs execute, nodes may update their replicas at different times or in different orders, which can cause replicated state to diverge and often results in erroneous application behavior. Our broad goal, first articulated in [Cheung et al. 2021], is to get the best of both worlds: allow developers to *write familiar sequential code* and use *program synthesis to lift* that code to an efficient distributed implementation.

A classic solution to bridge the gap between sequential and distributed semantics is to introduce **coordination**. Coordination allows all replicas to agree upon the order of execution; as a result, each replica can delay the application of early-arriving operations, which, if applied eagerly, would lead to divergence. Protocols for coordination (e.g. Paxos [Lamport 1998], Raft [Ongaro and Ousterhout 2014], and Zookeeper [Hunt et al. 2010]), offer a general-purpose solution to preserving sequential execution. However such approaches are prohibitively expensive for many applications—particularly at global scale [Crooks et al. 2016; Hellerstein and Alvaro 2020; Lakshman and Malik 2010]. Such coordination weakens the benefits of replication, as executing operations issued to single replicas now involves high-latency communication with other nodes. Thus, *avoiding* coordination has become a popular approach in the design of modern distributed systems [Cheung et al. 2021; DeCandia et al. 2007].

One of the most widely-adopted approaches for designing coordination-free programs is the use of **conflict-free replicated data types (CRDTs)** [Shapiro et al. 2011b]. Rather than relying on coordination to decide the order in which operations execute, CRDTs instead choose to limit their operations to only those which *commute*; if all replicas of a CRDT see the same set of commutative operations, then that CRDT will always *converge* to the same state, eliminating the threat of permanent replica divergence. As a result, the system will *eventually* reach a *consistent* state at each replica. For applications that can accommodate this *eventual consistency* [Vogels 2009], CRDTs enable coordination-free state replication. CRDT interpretations of traditional sequential datatypes, including shopping carts, maps, sets, and logs, have found wide adoption in both academia [Kleppmann and Beresford 2017; Weiss et al. 2009] and industry [Hoff 2014; Klophaus 2010; Roestenburg et al. 2016].

CRDTs provide a framework for coordination-free replication, but it is left to developers to design individual CRDTs that capture *application-specific* data models. For many, this is out of reach as ensuring convergence and semantic correctness is challenging even for experts [Kleppmann 2022]. Furthermore, specifying CRDTs is difficult, as the corresponding operations on sequential types are often *not* commutative. For example, one may wish to replicate a set that supports both insertions and removals, in which the order of insertions and removals is critical to the final state. The desire to use such data types has given rise to a class of CRDTs that *almost* match the behavior of a sequential data type. For example, remove operations will "appear" to evaluate before concurrent add operations in the Add-Wins Set CRDT, regardless of the order in which those operations arrive at each replica. These semantic differences make such CRDTs hard to verify [Gomes et al. 2017]. Moreover, these CRDTs require complex logic to capture the effects of operations while ensuring that replicas ultimately converge.

In this paper, we introduce Katara[1], an open-source system that automates the process of CRDT creation by leveraging **verified lifting** [Ahmad et al. 2019; Cheung et al. 2013; Kamil et al. 2016]. Using program synthesis techniques, we *lift* annotated implementations of sequential data types in languages like C/C++ to full implementations of nearly equivalent CRDTs. The sequential data types

[1] https://github.com/hydro-project/katara

being lifted appear as standard data structures in traditional software, can include constructs such as branching and loops, and do not need to come equipped with custom convergence properties. Users need only to annotate their sequential data types with a simple function that defines the order in which conflicting operations should *appear* to occur. For example, when lifting a set data structure, a user may choose to order removal operations before all concurrent addition operations, specifying the semantics of an Add-Wins Set. We then *automatically verify* synthesized CRDT candidates against a combination of the sequential semantics and user-specified conflict resolution policy. Crucially, such policies are easy to specify, requiring only a handful of lines in all our examples. By automating the process of verifying a candidate design, we can generate complete implementations of the CRDT's state, operations, and queries without any user intervention.

CRDT designs can be split into two categories: op-based CRDTs and state-based CRDTs. In this paper, we synthesize **state-based CRDTs (CvRDTs)**, as these can be deployed in more environments and can always be translated to op-based CRDTs if necessary [Shapiro et al. 2011a]. State-based CRDTs are defined by a datatype representing their state, and an associative, commutative, and idempotent (ACI) *merge function*, which determines how the states of replicas are combined to reach convergence. The state type and merge function together form a join semilattice, with the merge function serving as the join.

Early CRDT work involves complex state structures, but it was subsequently observed that CRDTs can be assembled via composition of simple join-semilattices on sets, integers, or Booleans [Conway et al. 2012; Wu et al. 2018]. We leverage this approach in the context of synthesis. By limiting the search space of state types to compositions of join-semilattices, we are able to achieve convergence—normally the most difficult property of CRDTs to verify—*entirely by construction.*

With this intuition, we introduce new algorithms for searching the space of possible CRDT implementations, including the state representation of the CRDT, operations defined on it, and the queries by which its state may be observed. This includes the design of grammars for runtime logic that we search with a Syntax-Guided Synthesis engine [Alur et al. 2013] and a parallelized enumerative search over compositions of semilattices for the state structure used within the CRDT. We apply multiple SMT encodings of our correctness conditions to quickly prune the CRDT search space and perform unbounded verification. Katara is able to automatically generate a variety of practical, provably correct CRDT designs for a wide range of specifications.

To summarize, we make the following contributions:

- We define a CRDT's correctness in terms of its operations and queries, and demonstrate how users can specify CRDTs by augmenting a sequential data type with lightweight ordering constraints that resolve conflicts between non-commutative operations (Section 3).
- We introduce an SMT encoding of our correctness conditions that enables automated verification of CRDTs against sequential data types with ordering constraints, along with a bounded variant that enables efficient pruning of the program search space (Section 4).
- We design a synthesis algorithm that efficiently searches semilattice compositions for the internal state of the CRDT, creates grammars for runtime components that guarantee convergence, and applies syntax-guided synthesis to generate both the core logic and invariants that prove correctness over unbounded executions (Section 5).
- We describe a practical implementation of Katara, including the details of how we automatically generate verification conditions from sequential data type implementations in C/C++ and optimize performance by synthesizing several candidate CRDTs in parallel (Section 6).
- We demonstrate how Katara can automatically lift sequential data types into practical CRDTs, and generate alternative designs with behavior equivalent to human-designed CRDTs in existing literature (Section 7).

2 MOTIVATING EXAMPLE

The distributed shopping carts problem, made popular by Amazon's Dynamo [DeCandia et al. 2007], is an essential business problem with a clever coordination-free solution. In the original formulation of this problem, the authors track a mapping of items to non-negative counts representing how many of that item are in the cart. Users can interact with the shopping cart by requesting insertions and removals of items. The cart can also be queried to determine the count of each item during a checkout procedure. The goal is to replicate a single shopping cart across many distributed nodes to improve fault tolerance, while ensuring eventual consistency so that the accumulated states on any node can be used to query the complete cart.

Let us focus on a simplified version of this problem, where each item can only be in the cart at most once—effectively simplifying the cart to a set of items. A developer without a distributed systems background could attempt to implement this as a replicated data type by having the insertions, removals, and queries all operate on a standard hash-set. However, if we deploy this in a distributed setting, we will immediately begin to see issues.

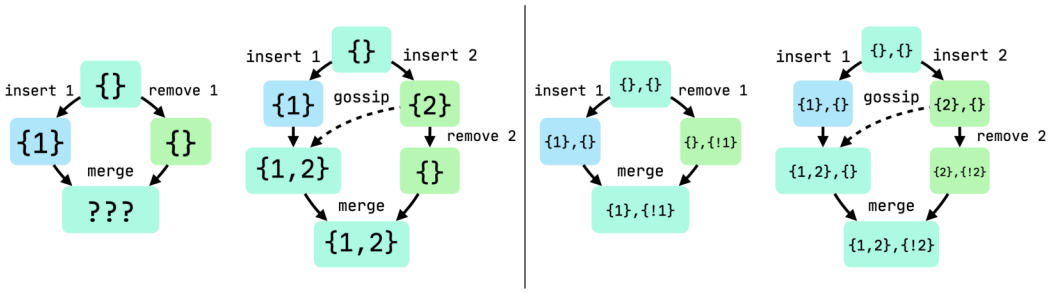

Fig. 1. Situations where a sequential type (left) has consistency issues that are resolved by a CRDT (right).

Consider the leftmost scenario in Figure 1, where a shopping cart is replicated across two nodes. When a user sends operations to add and remove the same item, these requests are distributed between the nodes. In this case, the node that receives the insert operation will add the item to its local set, but the node that receives the remove will treat it as a no-op because its local set is currently empty. We may wish to merge the state at these nodes via set union, but this would fail to preserve the locally-ineffective remove operation; it is unclear if this matches user expectations.

If we periodically share state via gossip [Demers et al. 1987], new situations emerge where consistency is disrupted. In the execution graph with gossip on the left of Figure 1, we see an execution where one node processes an insert, and the other processes an insert and then remove of the same element. Even if we merge via set union, we still end up with non-deterministic results depending on when gossip takes place. If the state of the right node is gossiped to the left after the insert but before the remove, the left node will merge the new value into its local state. Even after the remove is processed, merging the two states together will still result in the element 2 being present. But if the gossip were not to take place, the merged state would only have the element 1.

Situations like these have severe correctness implications, and it can be challenging to reason about the many ways a distributed system can break sequential assumptions built into a data type. Furthermore, it is challenging for developers to fix such correctness issues, because doing so involves reasoning about all possible interleavings of operations and their possible effects. Our work aims to tackle this issue by automatically synthesizing a CRDT, which satisfies the property

of *convergence* and can be safely replicated in a distributed cluster without requiring coordination. Let's explore how a user would synthesize a shopping cart CRDT with Katara.

Our synthesis algorithm takes two inputs: a sequential data type that defines the semantics of operations and queries the CRDT will support, and an ordering constraint that specifies how to resolve conflicts between non-commutative operations. We already have the first, since the user has implemented a sequential, single-node shopping cart. For the second, the non-commutative operations in the sequential type are inserts and removes, so the developer may decide that they want the CRDT to resolve conflicting operations by having the removes "win." This can be encoded as a simple pairwise ordering constraint (*opOrder*) that is passed into the synthesis algorithm.

Once Katara is given this specification, it searches potential state types and runtime logic that are both behaviorally correct and convergent. Along the way, our synthesis algorithm prunes out candidates by using bounded verification to quickly check correctness against short sequences of operations. Eventually, the synthesis engine will produce a provably correct CRDT. For our running example, we might get the CRDT in Figure 2 with an internal state of two sets $(s1, s2)$.

user input

```
set* init_state() { return set_create(); }

set* next_state(set* state, int add, int v) {
  if (add == 1) return set_insert(state, v);
  else return set_remove(state, v);
}

int query(set* state, int v) {
  return set_contains(state, v);
}
```

$opOrder(o_1, o_2)$: $o_{1,add} = 1 \vee o_{2,add} \neq 1$

synthesized design

crdt *ShoppingCart*
  initialState: $(\{\}, \{\})$
  merge $(a_1, a_2), (b_1, b_2)$
    return $(a_1 \cup b_1, a_2 \cup b_2)$
  operation $(s, add, value)$
    return $merge(s,$ **if** $add = 1$
      then
        $(\{value\}, \{\})$
      else
        $(\{\}, \{value\})$
      $)$
  query $((s_1, s_2), value)$
    return $value \in (s_1 \setminus s_2)$

Fig. 2. A user-provided sequential reference and the CRDT design synthesized by Katara.

Readers who are familiar with literature on CRDTs may recognize the implementation above as a Two-Phase Set [Shapiro et al. 2011a], one of the classic CRDTs that mimics the behavior of a set while guaranteeing convergence in distributed execution. This synthesized implementation is provably correct for the given sequential specification with the operation reordering, so we can deploy it in our distributed application without having to worry about manually proving complex CRDT properties. We can see on the right of Figure 1 that our CRDT now consistently handles the execution graphs that had conflicts with the naive implementation. Without any baked-in knowledge of existing CRDTs, Katara is able to automatically generate such implementations that were previously only designed by distributed systems researchers, and can even generate new undiscovered designs by searching the wider space of CRDT structures.

3 SPECIFYING CRDTS WITH SEQUENTIAL DATA TYPES

In verified lifting, a key piece of the puzzle is specifying correctness of the synthesized program in terms of the reference code. Past work has been focused on transpiling legacy functions into high-level DSLs, a domain in which correctness has a relatively simple definition: the synthesized logic must produce the same output as the original code for any valid input. When lifting CRDTs, however, our challenge is different: we are not just finding a function that produces the correct

input/output pairs, but rather finding a *stateful* data structure that produces the correct outputs for *unbounded sequences* of method invocations—including mutations and queries. In this section, we develop a formal model for sequential data types and CRDTs and introduce the correctness conditions for a CRDT to match a sequential specification.

We model sequential data types as a combination of two functions: a state transition ($st(s, o)$) that takes in the current state and an operation (a tuple of client-provided parameters) and returns an updated state, and a query function ($query(s, q)$) that takes the current state and a query (similarly, a tuple of client-provided parameters) and returns some data of any type. Sequential types also define an initial state ($initialState$) that is updated as operations are processed. This model can handle a wide range of sequential data types, including those that do not separate updates from queries, since independent operation/query endpoints can be combined into a single *st*/*query* function and we do not restrict the logic inside those functions.

On the CRDT side, we have a similar interface with one additional function to handle gossip from distributed nodes. In our discussion, we will distinguish the functions corresponding to a CRDT candidate from the sequential data type by attaching an asterisk to the CRDT names. Because the CRDT uses a different state type than the sequential structure, we also mark CRDT states with an asterisk (s^*). Just like before, we have an initial state ($initialState^*$), state transition ($st^*(s^*, o)$) returning a value of the CRDT state type, and query ($query^*(s^*, q)$) which together model how the CRDT processes requests over time. In addition, we introduce a merge function ($merge^*(s_1^*, s_2^*)$), which is used to merge a node's local state with gossip received from other nodes so that the replicated data converges.

CRDTs have two key components to their correctness: they must respond appropriately to operations and queries, and they must converge under eventual consistency. We guarantee the convergence by construction with grammar restrictions on the state (Section 5.1) and runtime logic (Section 5.2). In this section, we focus on specifying correct operations and queries by comparing the behavior of the CRDT to a reference sequential data type.

3.1 Specifying CRDTs with Operation Sequences

Because the sequential data type and the CRDT may use different internal state representations, we cannot directly compare instances of them by comparing their states. As a result, our definition of correctness must reason about the *user-observable behavior* of the data type over *unbounded sequences of interactions*. Both the sequential type and the CRDT have two components a user can interact with: the state transition and the query. Since queries are the only way for users to observe data, we want to ensure that after processing any sequence of user interactions, the sequential type and the CRDT respond identically to any query. Because queries do not modify the state, we can simplify this condition: a sequential type and CRDT are equivalent if both return the same result to an arbitrary query after processing an arbitrary sequence of operations.

But we have to go a step further to justify this correctness definition, since CRDTs can be executed in a distributed system. When we replicate the data type, operations will non-deterministically arrive at nodes in different orders. In addition, the use of gossip protocols in the cluster results in additional state updates when a node merges its local state with a state received from another node. As a result, instead of having a totally-ordered sequence of operations taking the initial state to the final one, we instead have a partial order—a directed acyclic graph—that captures the many different orderings of operations that could be seen by different replicas.

Thankfully, operations on a CRDT that satisfies an eventual consistency policy are commutative: the CRDT state depends only on the *unordered set* of observed operations. Even for CRDTs with causally consistent semantics, operations can be made commutative by extracting sources of causality such as timestamps into operation parameters, making them commutative *modulo* the

causality [Nagar and Jagannathan 2019]. Therefore, as long as the provided CRDT is convergent and has commutative operations, we can reduce verification of a distributed CRDT execution to the sequential case via an arbitrary flattening of the execution graph.

3.2 Resolving Commutativity with Operation Orderings

So far, our correctness conditions require strict equivalence of the sequential data type and the synthesized CRDT. However, the additional requirement of operator commutativity on the CRDT means that many sequential types with non-commutative operations, including common ones like sets and maps, cannot directly correspond to an equivalent CRDT but instead are mapped to many popular CRDT variants that make different semantic compromises.

In Katara, users can define these semantic adjustments through **operation orderings**, which loosen the correctness requirements to only verify sequences of operations following a specific ordering constraint. This approach, reminiscent of past work on CRDT specification [Burckhardt et al. 2014], minimizes user effort (by leveraging verified lifting) and enables automated verification (Section 4). Formally, a user can define a partial order $opOrder(o_1, o_2)$, which returns true when a call to o_2 is allowed to occur after a call to o_1.

As an example of the effect of this ordering, recall the shopping cart we lifted in the motivating example. In our sequential data type, inserts and removals do not commute, so no CRDT exists that strictly matches its semantics. However, we can resolve the conflict by introducing an ordering between the operations. If we specify that removes take place before inserts, we get a specification of a Grow-Only Set, which treats removes as no-ops. On the other hand, if we specify that removes take place after inserts, we get a specification of a Two-Phase Set, where elements can be inserted, then removed, but not inserted again.

Instead of having to consider all potential interleavings of operations in a distributed system, operation orderings make it possible for users to specify the distributed behavior in terms of a *sequential execution model*. This makes it possible for non-experts to use our synthesis algorithm, since they can reason about the operation ordering with the existing sequential data type.This general approach of ordering operations to resolve commutativity allows us to synthesize a wide variety of CRDTs by applying different orderings to simple sequential data types. Intuitively, ordering the non-commutative operations of the sequential type transforms the semantics to be effectively commutative, since any ordering of non-commutative operations will always be reordered into the same sequence by the *opOrder* constraint. Since the CRDTs we are verifying are already guaranteed to have commutative operations, it then suffices to verify correctness with sequences of operations that follow this order.

3.3 Operation Orderings with Time

A popular pattern when designing replicated data types is to place "distributed timestamps" on all operations, which makes it possible to introduce sequential semantics without losing convergence. For example, the Last-Writer-Wins Set is a classic CRDT that timestamps its operations; each operation supersedes any conflicting operations that have strictly earlier timestamps. Because distributed timestamps are only a partial order, there can be conflicts among operations that are incomparable in time, which are handled with remove- or add-wins semantics.

When a user provides a sequential data type and ordering specification, we allow them to enable a flag to introduce timestamps to each operation. This flag augments every operation with an integer timestamp o_t, which is computed by the local node at runtime using a source that can be mapped to an integer value. We use Lamport timestamps [Lamport 1978] as this source, which guarantees that causally ordered events will have accordingly ordered integer timestamps. We then augment the *opOrder* to order operations first by their timestamp, and then apply the user-defined

ordering on operations with the same timestamp:

$$opOrder(o_1, o_2) \triangleq (o_{1,t} < o_{2,t}) \vee ((o_{1,t} = o_{2,t}) \wedge opOrder_{orig}(o_1, o_2))$$

When we introduce time, we also must introduce constraints on the operations we consider in our correctness conditions to avoid degenerate cases with illegal timestamps. We do this through an additional user-defined function $opPrecondition(o)$, which checks that an operation is valid. When timestamps are enabled, we define the precondition as $opPrecondition(o) = o_t > 0$ to ensure the operations we check have valid timestamps. Users can also add constraints to this precondition based on domain knowledge, such as if an operation parameter will always be positive.

4 AUTOMATED CRDT VERIFICATION

Now, we must encode this formal definition of CRDT correctness to enable the automated verification of candidate CRDT designs. We tackle this by encoding correctness in SMT logic, which allows us to use solvers like Z3 [De Moura and Bjørner 2008] and CVC5 [Barbosa et al. 2022] to automatically prove correctness or find counterexamples. However, these solvers cannot directly reason about unbounded sequences of operations, so we must break down the correctness conditions into an inductive proof that reasons about individual state transitions.

4.1 State Equivalence

First, we focus on checking CRDT correctness without considering the ordering constraint. To build this inductive proof, we need a way to reason about the relationship between the states of the sequential data type and candidate CRDT after processing the same, arbitrary sequence of operations. To do this, we choose to relate the states of the CRDT and sequential data type implementations in the style of a bisimulation.

(1) $equivalent^*(initialState, initialState^*)$

(2) $\forall s, s^*, o : (equivalent^*(s, s^*) \wedge opPrecondition(o)) \implies equivalent^*(st(s, o), st^*(s^*, o))$

(3) $\forall s, s^*, q : equivalent^*(s, s^*) \implies query(s, q) = query^*(s^*, q)$

Fig. 3. The verification rules that constrain CRDT synthesis to preserve the source semantics

We encode this proof by introducing the **state equivalence** function, which relates the states of the sequential reference and synthesized CRDT. Formally, if the reference and synthesized data types are in equivalent states, then after both process an arbitrary sequence of operations they will return the same result to any query. Intuitively, the equivalence function describes which states of the sequential data type correspond to states of the CRDT that capture the same queryable knowledge. Furthermore, the equivalence function captures invariants about the CRDT that filter unreachable states from the verification conditions. With this function available as the inductive invariant of our bisimulation, we can now encode our correctness conditions in SMT logic.

We begin our conditions in Figure 3 with rule (1), that the initial states of the sequential data type and CRDT must be equivalent. Next, we build the inductive proof that carries equivalence all the way to the final query. We start by encoding the query constraint in rule (3), where we query the reference and synthesized implementations in equivalent states. For this condition, the query results are of the same type so we can directly check for equality. Intuitively, this rule requires equivalence to be a guarantee that queries on the two states return the same result. However, since equivalence deals with not only the current state but also queries on the future states, the

equivalence condition may need to be stronger. The condition that forces this strengthening is the inductive step of our proof in rule (2), which checks that if the two data types are in equivalent states, then after executing the same operation they should still be in equivalent states.

4.2 Enforcing Operation Orders with Invariant Synthesis

So far, our verification conditions ignore the presence of the user-defined operation ordering (*opOrder*), which specifies how the CRDT should handle conflicting non-commutative operations. In Katara, we implement two encodings of the constraints imposed by this ordering: one that supports unbounded verification but requires synthesizing additional invariants, and one that is only suitable for bounded verification but enables efficient exploration of the program space. In our end-to-end algorithm (Section 5.3), we use the bounded encoding first to quickly prune out candidate state structures.

To introduce the ordering constraint to the unbounded verification conditions, we take the approach of strengthening the inductive hypothesis by synthesizing additional invariants. The key insight in this approach is that CRDT states are accumulated by merging updates produced by each operation, so we can enforce orderings that use simple comparisons (such as equality or greater/less than) on the *accumulated state* instead of the individual operations in the history of the CRDT. For example, in the shopping cart scenario where inserts are ordered before removes, we know that an insert is in-order when the set of removed elements is empty.

Formally, we introduce another synthesized Boolean function $orderWithState^*(s^*, o)$, which returns true if the operation o satisfies the ordering constraint against the history of operations *implied* by the state of the CRDT. This function must be true when executing any operation in a correctly ordered sequence, which in turn ensures that the CRDT correctly handles all executions that satisfy the ordering constraint. By enforcing the user-provided ordering in terms of just the CRDT state, for which we already have inductive invariants, we are able to completely avoid the issue of separately reasoning about the history of operations that have been applied to the CRDT; we directly prove correctness in the unbounded case.

(1) $\forall o : equivalent^*(initialState, initialState^*) \wedge$
$(opPrecondition(o) \implies \boldsymbol{orderWithState^*(initialState^*, \mathbf{o})})$

(2) $\forall s, s^*, o_1, o_2 : (equivalent^*(s, s^*) \wedge opPrecondition(o_1) \wedge \boldsymbol{orderWithState^*(\mathbf{s}^*, \mathbf{o_1})})$
$\implies (equivalent^*(st(s, o_1), st^*(s^*, o_1)) \wedge ((opOrder(o_1, o_2) \wedge opPrecondition(o_2)) \implies$
$\boldsymbol{orderWithState^*(\mathbf{st}^*(\mathbf{s}^*, \mathbf{o_1}), \mathbf{o_2})}))$

(3) $\forall s, s^*, q : equivalent^*(s, s^*) \implies query(s, q) = query^*(s^*, q)$

Fig. 4. The verification rules updated to use a synthesized invariant for ordering constraints

To start, we update the base case to require that any operation executed in the initial state must be in-order. Then, we update the inductive step to enforce the correctness of $orderWithState^*$ on *all* operations in an ordered sequence, by extending rule (2) to reason about pairs of adjacent operations. We introduce a precondition that checks if the first operation being executed is in-order with the state, and enforce the transitive property that o_2 be in-order with the state after o_1 is processed if it is pairwise in-order after o_1. When these conditions are satisfied, we have a proof that our CRDT matches the sequential data type under the operation ordering for any unbounded execution.

4.3 Optimizing Verification with Quantified Queries

Because it is critical to verifying a CRDT candidate, the *equivalent** invariant must be synthesized alongside the other runtime logic. But with equivalence defined in terms of just the reference and synthesized states, synthesis can quickly become infeasible when the internal states involve large, unbounded structures such as sets and maps (Section 5.1), which would require the synthesizer to generate higher-order logic like reductions to compare the states. But we can significantly reduce this burden by noticing that beyond the inductive step, equivalence is only used as a precondition for checking that the sequential data type and CRDT return the same response for a *specific query*. Therefore, we can reduce the synthesis burden by introducing an additional parameter q to *equivalent** so that it is only responsible for checking that the states are observationally equivalent *for a given query*.

(1) $\forall o, \mathbf{q} : equivalent^*(initialState, initialState^*, \mathbf{q}) \wedge$
$(opPrecondition(o) \implies orderWithState^*(initialState^*, o))$

(2) $\forall s, s^*, o_1, o_2, \mathbf{q} : (equivalent^*(s, s^*, \mathbf{q}) \wedge opPrecondition(o_1) \wedge orderWithState^*(s^*, o_1))$
$\implies (equivalent^*(st(s, o_1), st^*(s^*, o_1), \mathbf{q}) \wedge ((opOrder(o_1, o_2) \wedge opPrecondition(o_2)) \implies$
$orderWithState^*(st^*(s^*, o_1), o_2)))$

(3) $\forall s, s^*, q : equivalent^*(s, s^*, \mathbf{q}) \implies query(s, q) = query^*(s^*, q)$

Fig. 5. The verification conditions with the additional query parameter for equivalence

By giving the equivalence function a specific query, the synthesized logic can now focus on comparing the parts of each state that are relevant to that query. For example, when synthesizing a CRDT that uses maps, this can result in significant simplifications like only checking one key. We update the verification conditions by adding a new quantifier for the query to rules (1) and (2). In rule (3), we simply pass the existing query variable to the *equivalent** function. We define these updated conditions in Figure 5. An additional optimization this enables in Section 5.2.1 is to move the postcondition of rule (3) into the structure of *equivalent**, which further reduces the burden on the synthesizer since it would otherwise have to discover this condition by itself.

4.4 Solution Pruning with Bounded Operation Logs

The unbounded verification conditions, while necessary to prove correctness for the CRDT, have a large performance impact on synthesis since they require both the CRDT and *orderWithState** to be synthesized simultaneously. To reduce this impact, we employ a two-phase synthesis approach where we synthesize the core logic of candidate CRDTs with verification conditions that check a bounded number of operations (and therefore do not require additional invariants), and separately synthesize the invariants to prove unbounded correctness.

There is one key modification to the base verification rules that we need to make: the state transition should only be checked when the operation being processed is in-order according the user-provided function. To encode this, we add an additional variable to the verification conditions (σ) that stores a bounded log of operations that have been processed. Because this operation log has a statically known bound, we can lower it to a fixed set of variables corresponding to each element of the list and avoid having to involve more complex theories. Note that the log cannot be used by any of the synthesized logic, since its only role is to aid verification.

$$\text{list in-order/valid} : lio(\sigma) \triangleq \forall i : (i < |\sigma|) \implies (opPrecondition(\sigma[i]) \wedge ((i < |\sigma| - 1) \implies opOrder(\sigma[i], \sigma[i+1])))$$

$$\text{list coherent} : lc(s^*, \sigma) \triangleq s^* = fold(\sigma, initialState^*, st^*)$$

(1) $\forall q : equivalent^*(initialState, initialState^*, q)$

(2) $\forall s, s^*, \boldsymbol{\sigma}, o, q : (equivalent^*(s, s^*, q) \wedge opPrecondition(o) \wedge \boldsymbol{lio(\sigma) \wedge lc(s^*, \sigma) \wedge opOrder(\sigma[|\sigma| - 1], o)}) \implies equivalent^*(st(s, o), st^*(s^*, o), q)$

(3) $\forall s, s^*, q : equivalent^*(s, s^*, q) \implies query(s, q) = query^*(s^*, q)$

Fig. 6. The verification rules updated to use operation logs for bounded ordering constraints

Then, we update the state transition verification rule to add a precondition that the operation log is in-order and coherent with the state of the CRDT. First, we check that every pair of operations in the log are in-order, since the operation log is a quantified variable in the SMT encoding and may have out of order values. Then, we verify that the synthesized state equals the result of folding over the log with the synthesized state transition. Since the operation log is bounded, this collapses into a bounded number of state transitions and can be efficiently verified by an SMT solver. Finally, we add a condition that the operation being applied in rule (2) is in-order with the existing log, which can be checked by comparing it against the last operation (since the ordering constraint is transitive).

Note that we do not need to introduce the ordering and coherence preconditions to the query verification conditions in rule (3) even though that rule operates on arbitrary input states. Because we still synthesize the *equivalent** function to relate the CRDT and sequential data type, we do not need to constrain *how* we reach the synthesized state being evaluated, just that any instance of the sequential reference deemed equivalent will return the same response to the query. It is left to the synthesizer to introduce any invariants necessary to avoid checking queries on unreachable states. Together, these rule modifications are summarized in Figure 6. With bounded operation logs, this encoding enables efficient synthesis of the state transition and query functions.

5 CRDT SYNTHESIS ALGORITHM

Now that we have a formal specification of correctness that can be verified by an SMT solver, we are ready to define the synthesis algorithm for CRDT implementations. Our end-to-end algorithm requires only two pieces of user input: the sequential data type written in a standard imperative language and the operation ordering that resolves conflicts. Our synthesis algorithm optionally takes a set of Boolean flags that enable synthesis of advanced designs, such as those that use timestamps (discussed in Section 3.3) or have non-idempotent operations (which we explore later in this section).

There are four core components to synthesize: the type of the internal state, the initial state, the state transition function, and the query function. We must also synthesize the *equivalent* and *orderWithState* invariants from the previous section to enable verification. As discussed before, the synthesized CRDT may use a completely different state structure than the source, which adds a new layer of complexity because the choice of state type affects the search space for each synthesized function. Therefore, our synthesis algorithm uses multiple phases to generate candidates of runtime logic for a range of potential state types.

In Section 3, we explained that our verification conditions only check the user-observable behavior of the CRDT, but do not verify that the CRDT implementation meets the convergence properties. Instead of checking these properties through verification conditions [Nagar and Jagannathan 2019], we craft our CRDTs in a way that satisfies these properties *by construction*. Inspired by past work on designing coordination-free distributed systems [Conway et al. 2012; Wu et al. 2018], we synthesize CRDTs that use **semilattice compositions** for their internal state, which makes it straightforward to enforce monotonicity and commutativity since these are properties of the semilattice join.

5.1 State Synthesis

To explore candidate state structures for the CRDT, we use the classic synthesis approach of defining a grammar and iteratively processing deeper structures. Because we focus on compositions of semilattices, our grammar consists of simple rules for primitives, sets, maps, and tuples.

For primitive types, we include semilattice definitions based on Booleans and integers, which are sufficient to lift a wide variety of sequential data types. For Booleans, we have the `OrBool` lattice, which is a Boolean that has $\bot = false$ and is merged with $\vee$. For integers, we provide the `MaxInt` semilattice, which merges integers by taking the maximum. Beyond the primitives, we include a semilattice definition for `Set<T>`, which can have a non-lattice type `T` for elements; the only constraint on `T` is that it supports equality.

Our lattice definitions for composite data structures are more complex. First, we offer the `LexicalProduct<A, B>` semilattice, where A and B are themselves semilattices. In this semilattice, the first element has priority over the second when determining the ordering of two instances. This type is especially useful for CRDTs that use timestamps to have recent operations override older ones, but need to perform a merge over the underlying values when the effects of concurrent operations are combined. We define the lattice join for `LexicalProduct` as:

$$(a_1, b_1) \sqcup (a_2, b_2) = \begin{cases} (a_1, b_1) & a_1 > a_2 \\ (a_2, b_2) & a_2 > a_1 \\ (a_1 \sqcup a_2, b_1 \sqcup b_2) & \text{otherwise} \end{cases}$$

This definition respects the lattice axioms of associativity, commutativity, and idempotence. Furthermore, these tuples can be nested to form tuples of arbitrary arity. We also support the `FreeTuple<A,B>` lattice, which simply joins elements pairwise (i.e. $(a_1, b_1) \sqcup (a_2, b_2) = (a_1 \sqcup a_2, b_1 \sqcup b_2)$) and can similarly be nested to form tuples of arbitrary arity.

In some cases we may not know the desired arity of a `FreeTuple` in advance, or we may not need all the "fields" of such a tuple in a given execution. To address this, we offer a `Map<K, V>` semilattice, where `K` can be any type that supports equality, and `V` is a semilattice. Our maps support common operations such as insertions with the same semantics as regular maps, except when inserting keys that are already present in the map. Instead of overwriting the value, we use the lattice join for the value type to combine the existing value with the one being inserted. This carries over to our definition of the lattice join for maps themselves, where we insert the entries of both maps, with keys that are present in both maps having their values merged according to their join:

$$m_1 \sqcup m_2 = \{k_i : \begin{cases} m_1[i] & (k_i \in m_1) \wedge (k_i \notin m_2) \\ m_2[i] & (k_i \notin m_1) \wedge (k_i \in m_2)\ ) \\ m_1[i] \sqcup m_2[i] & (k_i \in m_1) \wedge (k_i \in m_2) \end{cases}\}$$

Again, this definition respects the standard lattice axioms. Given these semilattice types, we can construct the grammar in Figure 7 that defines the space of compositions to explore. Our grammar

covers a large space of semantics, since the available types encode core capabilities such as free and lexicographically-ordered semilattice products (via maps and tupling) and general semilattice representations (sets). In our end-to-end synthesis algorithm, we explore types in this grammar with iteratively increasing depth bounds and attempt to synthesize the runtime component of the CRDT for each one. Note that we only include `FreeTuple` in the top-level *latticeList* for CRDTs that need multiple semilattices in their state.

$$\langle latticeList \rangle ::= \langle latticeType \rangle \mid \text{FreeTuple}(\langle latticeType \rangle, \langle latticeList \rangle)$$

$$\begin{aligned}\langle latticeType \rangle ::= {} & \text{OrBool} \mid \text{NegBool} \mid \text{MaxInt} \\ \mid {} & \text{Set}(\langle type \rangle) \mid \text{Map}(\langle type \rangle, \langle latticeType \rangle) \\ \mid {} & \text{LexicalProduct}(\langle latticeType \rangle, \langle latticeType \rangle)\end{aligned}$$

$$\langle type \rangle ::= \text{Bool} \mid \text{Int}$$

Fig. 7. The grammar defining compositions of semilattices we explore during synthesis.

5.2 Runtime Synthesis

With our state structure selected, we can now move on to synthesizing the runtime logic. Our algorithm for runtime synthesis proceeds in two phases: a first step that synthesizes the core logic with the bounded operation log verification conditions, and a second that synthesizes the additional invariants required for unbounded verification. By using a two-phase approach, we are able to significantly improve the end-to-end synthesis performance of our algorithm by pruning out state structure candidates for which no runtime implementation satisfies even the bounded conditions. In addition, this approach reduces the number of invariants that must be synthesized simultaneously with the CRDT logic, which further improves efficiency.

5.2.1 Core Logic Synthesis. We derive significant power from our choice to implement the internal state of the CRDT via a semilattice. First, we observe that, as lattice join is compositional, we can define the merge function as the lattice join on the internal state; this in turn is derived directly from the state's constituent lattices. Next, we observe that we can also implement *operations* in terms of this lattice join: we define our state transition as $st^*(s^*, o) = merge^*(s^*, f^*(o))$, where f^*, which returns a lattice value of the same type as s^*, is the function that we actually synthesize. This choice grants us monotonicity, commutativity, associativity, and idempotence entirely for free, derived from the lattice join ($merge^*$) itself.

Along with the state transition, we synthesize the query function that is used to pull information out of the CRDT. There are no convergence restrictions on the query since it does not mutate the state, leaving only the sequential reference as a source of constraints on its synthesis. As a result, we do not need to craft the query function in any special way, and can let the synthesis engine drive the search of the query logic.

To support the inductive step of the verification conditions, we must synthesize the equivalence function. This function has two intuitive roles: (1) a *cross-state relation* that identifies which states of the sequential data type and the CRDT are observationally equivalent, and (2) a *CRDT state invariant* that is needed to strengthen the inductive hypothesis of the correctness proof. Following this intuition, we split the synthesis of the equivalence function into components for each role. As discussed when we introduced the query parameter to $equivalent^*$ in Section 4.1, we seed the equivalence function with a check that both states respond identically to the given query. This means that our equivalence function has the form $equivalent^*(s, s^*, q) \triangleq query(s, q) = query^*(s^*, q) \wedge relation^*(s, s^*) \wedge invariant^*(s^*)$, where $relation^*$ and $invariant^*$ are synthesized.

Because the embedded query comparison already filters out most states that immediately return different query responses, we can improve synthesis performance with a heuristic that bounds the maximum expression depth of *relation** to one less than the other functions. Even with this optimization, we can synthesize complex *relation** logic when necessary because the depth is iteratively increased. With the bounded operation log encoding, the *invariant** component is unnecessary because we know that the CRDT state can be reached through the explicit log of operations. We will revisit the invariant in Section 5.2.4, when we synthesize the CRDT using the encoding for unbounded correctness that does not use a log.

$\forall T, U, O$

$\langle bool \rangle ::=$ false | true
| $\langle bool \rangle \wedge \langle bool \rangle \mid \langle bool \rangle \vee \langle bool \rangle$
| $\neg \langle bool \rangle$
| $\langle int \rangle > \langle int \rangle \mid \langle int \rangle \geq \langle int \rangle$
| $\langle T \rangle = \langle T \rangle$
| $\langle T \rangle \in \langle Set(T) \rangle \mid \langle Set(T) \rangle \subset \langle Set(T) \rangle$

$\langle int \rangle ::= 0 \mid 1 \mid \langle int \rangle + \langle int \rangle \mid \langle int \rangle - \langle int \rangle$
| constants in the sequential source

$\langle Set(T) \rangle ::= \{\} \mid \{\langle T \rangle\}$
| $\langle Set(T) \rangle \cup \{\langle T \rangle\} \mid \langle Set(T) \rangle \cup \langle Set(T) \rangle$
| $\langle Set(T) \rangle \setminus \langle Set(T) \rangle$

$\langle Map(T, U) \rangle ::= \{\} \mid \{\langle T \rangle : \langle U \rangle\}$
| $\langle Map(T, U) \rangle \cup \langle Map(T, U) \rangle$

$\langle U \rangle ::= \langle Map(T, U) \rangle[\langle T \rangle, \text{default}=\langle U \rangle]$
| $\langle Tuple(U, T) \rangle[0] \mid \langle Tuple(T, U) \rangle[1]$
| input of type U

if top-level or U is not a Set or Map:
$\langle U \rangle ::=$ if $\langle bool \rangle$ then $\langle U \rangle$ else $\langle U \rangle$

Fig. 8. The core grammar used to synthesize the state transition and query functions.

The synthesized components of the state transition, query, and equivalence functions all use a common core grammar. Similar to past program synthesis work, we generate the grammar in Figure 8 based on the type constraints of supported operations and bound it by an iteratively increased depth, discussed further in Section 5.3. Our grammar features the core set of operations available on the types we support in our system, such as arithmetic, Boolean logic, and set/map operations. In addition, we include conditionals in our grammar to support branching inside the synthesized logic. Because branches that emit complex types such as sets or maps are expensive to synthesize, we restrict those to the top-level of the expression and seed the condition with any Boolean inputs and equality comparisons for integer inputs.

The astute reader may notice that this grammar does not enforce any of the ACI properties. But this is not a problem! Recall that we are synthesizing a function $f^*(o)$ that returns a lattice value to be passed into *merge**. Therefore, even though some operations in this grammar are not idempotent or commutative, the overall state transition function st^* remains associative, commutative and idempotent by construction. The semantics of the operations in our language are largely standard, and we lower the operations directly to the corresponding logic in the SMT solver when possible.

Finally, we synthesize the initial state using a shallow grammar of constructors and relevant constants for each type. We include small integer literals, Boolean constants, and empty instances of sets and maps. In cases in which $\perp$ is defined, the initial state is often synthesized to just be the bottom value of the lattice, but occasionally we want the synthesizer to pick an alternate value to handle queries in the initial state. For example, when synthesizing a Boolean register where concurrent enables are ordered *after* disables, the natural lattice to represent the flag's state is a `LexicalPair<ClockInt, OrBool>` with $\perp = (0, false)$, but we need the initial state to be $(0, true)$ if the sequential data type starts in a enabled state. By synthesizing this value instead of fixing it

to $\bot$, we can synthesize CRDT designs over semilattices that do not define bottom, or where the initial state starts higher in the semilattice order.

5.2.2 Synthesizing Non-Idempotent Operations. So far, the state transitions we can synthesize are always idempotent, which is not a requirement of CRDTs in general and prevents us from synthesizing designs such as counters. To resolve this limitation, we use a common trick in replicated distributed systems and relax the idempotence constraint while ensuring that certain state can only have a single writer via constraints on the state transition grammar. We introduce **node IDs**, which are unique integer identifiers for each node in the cluster that can be used as map keys to separate portions of the state that are tied to each node. With this separation of writable state, we can synthesize non-idempotent operations that update portions of the state that only the current node can write.

Users can introduce node IDs to the synthesis pipeline by enabling a single Boolean flag when the sequential data type has non-idempotent operations. Because synthesizing CRDTs with non-idempotent operators introduces additional variables and a larger grammar, which can impact performance, the flag is disabled by default and must be explicitly enabled by the user based on their knowledge of the sequential data type. In future work, we hope to automate this process by analyzing the sequential reference to automatically detect non-idempotence.

Enabling non-idempotent operators affects two components of the synthesis algorithm: the structure of the synthesized functions and the grammar used for runtime logic. The synthesized component of the state transition, which previously could only read the operation arguments to ensure idempotence and commutativity, is expanded to have access to the CRDT state as well as the current node ID. As a result, we must now synthesize a function with the form $f^*(o, s^*, \text{currentNodeID})$.

To synthesize CRDT logic that uses node IDs, we add a production rule so that the state transition can read from portions of the state that are owned by the current node, which are the values of maps keyed by a node ID. Similarly, we add a rule that allows the state transition to update portions of the state that the current node owns, by allowing insertions keyed by the current node ID. Finally, we introduce rules to the query grammar for performing reductions over the values of maps keyed by node IDs, which makes it possible to combine the state of each node into a global response to queries. We detail these additional grammar elements in Figure 9.

for the state transition:
$\forall V$
$\langle V \rangle$ ::= $\langle$*Map(NodeID,V)*$\rangle$[currentNodeID, default=$\langle V \rangle$]

$\langle$*Map(NodeID,V)*$\rangle$ ::= $\langle$*Map(NodeID,V)*$\rangle \sqcup$ {currentNodeID: $\langle V \rangle$}

for queries:
$\langle$*Int*$\rangle$ ::= reduce(values($\langle$*Map(NodeID,Int)*$\rangle$), $\lambda a.\lambda b.a + b$, 0)

$\langle$*Bool*$\rangle$ ::= reduce(values($\langle$*Map(NodeID,Bool)*$\rangle$), $\lambda a.\lambda b.a \vee b$, $false$)
| reduce(values($\langle$*Map(NodeID,Bool)*$\rangle$), $\lambda a.\lambda b.a \wedge b$, $true$)

Fig. 9. The additional production rules required to support non-idempotent operations.

Although these changes to the construction of the state transition may allow it to be non-idempotent (and potentially non-commutative), the synthesized CRDT remains convergent because the only requirement for state-based CRDTs is that the merge function agrees with the state transition. Because our state transition still performs a lattice join with the previous state at the

top level, and the non-idempotence/commutativity is restricted to portions of state, each owned by an individual node in the cluster, the merge function remains correct since a node can never receive new information about the portions of state it owns through gossip from other nodes.

5.2.3 Pruning Grammars with Specialized Types. While shallow instantiations of these grammars are sufficient to synthesize simple CRDTs, such as grow-only sets, they quickly grow to infeasible sizes when the state structure involves a larger number of nested data structures. Much of the grammar expansion comes from a conflation of types that can have distinct semantic meanings, resulting in production rules like arithmetic and comparison operations being unnecessarily introduced.

```
set* init_state() { return set_create(); }

set* st(set* s, int add, int v) { ... }

int query(set* state, int v) { ... }
```

```
stateTypeHint = Set(OpaqueInt())
opArgTypeHint = [EnumInt(), OpaqueInt()]
queryArgTypeHint = [OpaqueInt()]
queryRetTypeHint = EnumInt()
```

Fig. 10. An example of how a sequential data type is annotated with specialized types.

To resolve this, we introduce *specialized* integer types, which represent distinct interpretations of integer values. In Katara, we have `OpaqueInt`, which represents an abstract value that does not support arithmetic, `ClockInt`, which represents a positive timestamp that only supports comparison operations, and `EnumInt`, which represents values that only support equality. Users can then annotate the functions in their sequential data types, as shown in Figure 10, to mark types in the state and operation/query functions that conform to these specialized alternatives. When timestamps are enabled by the user to define richer operation orderings, we automatically add the necessary `ClockInt` annotations for those values.

By using distinct types for integer inputs, we can avoid searching expressions that, for example, compare timestamps to opaque values. We define grammar rules for these types in Figure 11. These types are also added to the state structure grammar, but for brevity we omit the changes here.

$$\begin{aligned}
&\langle bool \rangle ::= \langle opaque \rangle > \langle opaque \rangle \mid \langle opaque \rangle \geq \langle opaque \rangle \mid \langle opaque \rangle = \langle opaque \rangle \\
&\quad \mid \quad \langle clock \rangle > \langle clock \rangle \mid \langle clock \rangle \geq \langle clock \rangle \mid \langle clock \rangle = \langle clock \rangle \\
&\quad \mid \quad \langle enum \rangle = \langle enum \rangle
\end{aligned}$$

$$\langle clock \rangle ::= 0$$

$$\langle enum \rangle ::= 0 \mid 1 \mid \text{constants in the sequential source}$$

$$\begin{aligned}
&\forall T, U \\
&\langle U \rangle ::= \text{reduce}(\text{values}(\langle Map(T,U) \rangle), \lambda a.\lambda b.a \sqcup b, \bot)
\end{aligned}$$

Fig. 11. The production rules for specialized integer types and semilattice reductions.

With the grammars defined for all three functions, we can apply a syntax-guided synthesis algorithm to explore the space of CRDT implementations and use the bounded operation log verification conditions to automatically verify candidates using an SMT solver. The bounds used in this phase start at very small values but are incrementally increased based on feedback from later phases of the synthesis algorithm, which we discuss in further detail in Section 5.3.

5.2.4 Invariant Synthesis for Unbounded Verification. After the first synthesis phase produces a CRDT design that passes bounded-log verification, we must synthesize additional invariants to check the CRDT against the unbounded verification conditions. We must re-synthesize the equivalence function with the CRDT state invariant included, since the unbounded conditions depend on the invariant to exclude unreachable CRDT states. In addition, we synthesize the *orderWithState** function so that the unbounded conditions can reason about operation orderings.

The CRDT state invariant only has access to the CRDT state, which helps reduce the size of the grammar generated. We seed the invariant with an explicit condition that checks if the state is valid according to the relevant semilattice definitions. Each semilattice in our state grammar comes with logic for checking validity, such as that the integer values for clocks are at least zero. By automatically including these checks, we further reduce the burden on the synthesizer to discover properties needed for the inductive proof. The rest of the invariant is synthesized using the same type-based grammar as the other functions. Note that we do not need to synthesize the relation component of equivalence, since that was already synthesized in the bounded-log phase.

Synthesizing *orderWithState** is a bit more complex. Since the role of this function is to determine whether an operation is in-order while only having access to the CRDT state, this function often needs to combine information from large portions of the state rather than just manipulating data associated with specific keys. For example, when synthesizing a CRDT that uses clocks to order operations, *orderWithState** will need to check that the timestamp of the given operation is greater than all existing timestamps in the state. But it is challenging for syntax-guided synthesis engines to reason about arbitrary reductions, so we must reduce the complexity of the grammar.

We tackle this by noting that reductions (such as collecting the highest timestamp) use the semilattice join of the type being accumulated. This has intuitive backing as well, since we can check if a single value is at least as high in the semilattice order as several others through a single comparison against the semilattice join over those values. Based on this observation, we add a rule in Figure 11 to compute reductions using the lattice join for all relevant lattice values in the state. Note that we support reductions over map values, which are of a type in <*latticeType*>, but not sets because their elements may not be semilattices.

With these additional production rules, we can synthesize *invariant** and *orderWithState** for the candidate CRDT. With the invariant grammars configured and the existing *st** and *query** functions from the previous phase, we return to the synthesis engine with the unbounded verification conditions. At this point, we are verifying all scenarios the CRDT is expected to correctly handle, so if we successfully synthesize the invariants we have a provably correct CRDT design!

5.3 End-to-End Synthesis Algorithm

Now that we have the search space for state structures and runtime logic defined, we can synthesize the entire CRDT from scratch by simultaneously exploring both spaces. In our end-to-end algorithm, we apply multiple logic synthesis phases and verification modes to create provably correct CRDTs while also pruning the program space early in the synthesis algorithm.

At the top level of the algorithm, we iterate over candidate state structures generated from the grammar of semilattice compositions, bounded to the same *depth* as the runtime logic. For each of these, we then generate the appropriate runtime logic grammars and perform synthesis with the bounded-log verification conditions (with an initial $logBound = 2$). If we fail to synthesize, we can eliminate the candidate state structure from consideration, since there is no synthesizable logic even when the verification is relaxed.

If we successfully synthesize, we can move on to synthesizing the additional invariants for unbounded verification. We combine the synthesized code from the previous phase with the grammars for *invariant** and *orderWithState**, and call out to the synthesis engine again. If we

```
function search(ref, opOrder)
    for depth ← (2..∞) do
        for s ← semilatticeCompositions(depth) do
            logBound ← 2
            p2Depth ← depth
            loop
                p1Synth ← synthBoundedLog(ref, s, opOrder, depth, logBound)
                if p1Synth = unsat then
                    break
                p2Synth ← synthUnbounded(ref, s, opOrder, p2Depth, p1Synth)
                if p2Synth = unsat then
                    logBound ← logBound + 1
                    p2Depth ← p2Depth + 1
                else
                    return p2Synth
```

Algorithm 1: The end-to-end algorithm for synthesizing a CRDT from scratch.

fail to synthesize at this point, it means that either the bounded-log phase returned a buggy implementation or the grammar for invariants was too small. To address this, we return to the bounded-log phase and increment the operation log bound for verification and the grammar depth (*p2Depth*) for invariants. If we successfully synthesize the invariants, we have a provably correct CRDT that we can return to the user. We summarize this process in Algorithm 1.

6 IMPLEMENTATION

We implement Katara using an extended version of the framework in Casper [Ahmad and Cheung 2018], which allows us to automatically extract sequential data types written in C and C++ by first compiling them to LLVM and analyzing the IR to generate the equivalent SMT logic. Our implementation also includes wrappers around the Rosette synthesis engine [Torlak and Bodik 2013] and CVC5 solver [Barbosa et al. 2022], which we use to perform synthesis and verification.

6.1 Supported Language Features

To synthesize a CRDT, Katara must first extract the semantics of the sequential data type provided by the user. To ensure that the logic implemented by the user can be accurately translated into the SMT logic used for verification, we define a space of programs that can be safely handled. Katara can handle basic LLVM operations, branches, integer/Boolean primitives, and list/set/map types.

Our analysis can accurately handle primitive types such as integers and Booleans, along with the corresponding arithmetic and logical operations on them. In addition, we provide a set of APIs for lists, sets, and maps that users can build on in their sequential data type. Our analysis automatically recognizes uses of these specialized APIs and lowers them to the corresponding SMT theory. Our framework offers a modular approach to defining the semantics of these types, so it is straightforward to add support for richer data structures such as stacks.

In addition to analyzing the types and operations on them, our framework can extract branches found in the LLVM IR to conditionals in the generated SMT logic. Katara generates separate specifications of each basic block found in a function, and links them together to define the function as a whole. This approach allows us to handle nested conditionals and early returns without needing additional logic for these cases. In addition to branches, our analysis also handles user-defined functions by inlining them into the top-level function that is lifted.

6.2 Bounded Data Structure Verification

When synthesizing with Rosette, we face a limitation that the size of the symbolic state must be a constant, which means that we cannot define verification conditions that operate over unbounded data structures such as lists or sets. To address this limitation, we bound the size of these data structures to a fixed value while performing synthesis. After Rosette returns us a successfully synthesized CRDT, we then pass the result to CVC5, which can perform verification with unbounded data structures when a theory is defined for their behavior.

CVC5 natively supports a theory of sets, and we provide our own set of axioms that define tuples. When maps—which have not yet been modeled in CVC5—are involved in the synthesized CRDT, we fall back to using Rosette for verification with a large bound for the data structures. We hope to improve this in future work by providing a set of complete axioms that enable the solver to reason about unbounded map instances. In the meantime, the bounds we use for the fallback are sufficiently large to consistently produce correct synthesized results.

6.3 Parallel State Structure Exploration

Both Rosette, which uses Z3 under the hood, and CVC5 are single-threaded. If we were to naively implement the end-to-end algorithm, we would underutilize the multi-core capabilities of modern systems. But because we control the search of CRDT state structures, and the logic synthesis for each structure candidate is independent of the others, we can drastically speed up the synthesis algorithm by parallelizing across state structures.

Katara allows users to configure the number of state structures to synthesize logic for in parallel. Based on this parameter, we then instantiate a thread pool and spawn the logic synthesis algorithm for candidate structures on free threads. Our implementation simply returns the first successfully synthesized CRDT from any thread. By exploring more state structures and avoiding situations where synthesis is blocked on a state candidate that is particularly difficult to synthesize logic for, this technique drastically improves the end-to-end synthesis performance.

7 EVALUATION

In our evaluation, we explore the capability of Katara to *correctly* and *efficiently* synthesize CRDTs for a variety of sequential data type and operation ordering specifications. We focus on answering the following research questions:

- **RQ1**: Can Katara produce practical CRDTs based on specifications sourced from literature on coordination-avoidance?
- **RQ2**: What is the effect of pruning structures with bounded-log verification on the overall synthesis performance?
- **RQ3**: Is the grammar of lattice composition sufficiently rich to produce CRDT designs that differ from the canonical implementations in the literature?

7.1 RQ1: Synthesizing Practical CRDTs

We begin by evaluating the ability of our synthesis algorithm to produce correct CRDTs from scratch for a variety of user-provided specifications. We sourced several sequential data types and operation orderings, summarized in Table 1, from existing literature on human-designed CRDTs [Shapiro et al. 2011a] and coordination-avoidance [De Porre et al. 2021]. For each of the benchmarks, we created a minimal implementation of the sequential type in C based on the specifications provided by the source and encoded the operation ordering using the IR provided by the synthesis system. All benchmarks were conducted on a AMD Ryzen 9 3900X processor with 12C/24T and 48 GB of

memory, with our implementation configured to use up to 12 threads. We use LLVM 11 to compile and analyze the sequential types, as well as the latest versions of Rosette (4.1) and CVC5 (1.0.2).

Table 1. The set of CRDT specifications used to evaluate our synthesis algorithm.

Benchmark	Source	Specification Size	Timestamps	Non-Idempotent
Grow-Only Counter	Shapiro	21 LoC		✓
General Counter	Shapiro	20 LoC		✓
Enable-Wins Flag	De Porre	21 LoC	✓	
Disable-Wins Flag	De Porre	21 LoC	✓	
Last-Writer-Wins Register	Shapiro	16 LoC	✓	
Grow-Only Set	Shapiro	24 LoC		
Two-Phase Set	Shapiro	24 LoC		
Add-Wins Set	De Porre	24 LoC	✓	
Remove-Wins Set	De Porre	24 LoC	✓	

Our overall approach is designed to require minimal user intervention to produce a practical CRDT. As a proxy for this goal, we measured the amount of code required for a user to specify both the sequential data type and the operation ordering that specifies the synthesized CRDT. For all our benchmarks, both of these components can be declared within 25 lines of C (for the data type) and Python (for the operation ordering). The Boolean flags to enable timestamps and non-idempotence are also provided to the system through a Python API. Most of the lines of specification code are for the sequential data type, which a developer using Katara will likely already have. The operation orderings, which are specific to Katara, could all be defined in 4 LoC or less of integer comparisons.

Table 2. The performance of synthesizing CRDTs for the benchmark specifications with Katara.

Benchmark	Synthesis Time (default)	Synthesis Time (no pruning)	Synthesized State Type
Grow-Only Counter	1m 46s ± 0.7s	52s ± 0.3s	`Map<NodeID, MaxInt<Int>>`
General Counter	11m 3s ± 4s	13m 21s ± 4s	`FreeTuple<Map<NodeID, MaxInt<Int>>, Map<NodeID, MaxInt<Int>>>`
Enable-Wins Flag	2m 4s ± 1s	2m 53s ± 8s	`LexicalProduct<MaxInt<ClockInt>, OrBool>`
Disable-Wins Flag	1m 46s ± 3s	3m 4s ± 2s	`LexicalProduct<MaxInt<ClockInt>, OrBool>`
Last-Writer-Wins Register	26s ± 0.2s	18s ± 0.8s	`LexicalProduct<MaxInt<ClockInt>, MaxInt<Opaque>>`
Grow-Only Set	30s ± 0.2s	23s ± 0.1s	`Set<Opaque>`
Two-Phase Set	58s ± 0.6s	1m 5s ± 0.8s	`Map<Opaque, OrBool>`
Add-Wins Set	21m 17s ± 11s	1hr 58m ± 1m	`FreeTuple<Map<Opaque, MaxInt<ClockInt>>, Map<Opaque, MaxInt<ClockInt>>>`
Remove-Wins Set	18m 40s ± 14s	1hr 52m ± 1m	`FreeTuple<Map<Opaque, MaxInt<ClockInt>>, Map<Opaque, MaxInt<ClockInt>>>`

When run with our collection of benchmarks, our synthesis algorithm is able to successfully generate designs that conform to all of the specifications, and it identifies the inductive invariants necessary to prove correctness of each CRDT over unbounded executions. We list the average time required to synthesize each CRDT (along with standard deviations) and the state structure of the

synthesized result in Table 2. Simpler CRDTs, such as the Grow-Only/Two-Phase Set and LWW-Register, can be synthesized by Katara in a matter of seconds. More complex CRDTs, especially those that use timestamps to order operations such as the Add/Remove-Wins Set, can take on the order of tens of minutes to synthesize. These performance measurements indicate that the composition of multiple synthesis phases allows for many types of CRDTs to be synthesized in a reasonable amount of time.

crdt *AddWinsSet*
  initialState: $(\{\}, \{\})$
  operation $(s, add, value, time)$
    return $s \sqcup$ **if** $add = 1$ **then**
      $(\{value : time\}, \{\})$
    else
      $(\{\}, \{value : time\})$
  query $((s_1, s_2), v)$
    $t_1 = s_1[v, default = 0]$
    $t_2 = s_2[v, default = 0]$
    return $t_1 \geq t_2 \wedge t_1 > 0$

crdt *GeneralCounter*
  initialState: $(\{\}, \{\})$
  operation $((s_1, s_2), inc, nodeID)$
    $cur_1 = s_1[nodeID, default = 0]$
    $cur_2 = s_2[nodeID, default = 0]$
    return $(s_1, s_2) \sqcup$ **if** $inc = 1$ **then**
      $(\{nodeID : cur_1 + 1\}, \{\})$
    else
      $(\{\}, \{nodeID : cur_1 + 1\})$
  query $((s_1, s_2))$
    $r_1 = reduce(values(s_1), \lambda a.\lambda b.a + b, 0)$
    $r_2 = reduce(values(s_2), \lambda a.\lambda b.a + b, 0)$
    return $r_1 - r_2$

Fig. 12. The Add-Wins Set and General Counter CRDTs our synthesis algorithm is able to produce.

Our synthesis results also show the algorithm discovering a variety of CRDT design techniques without any baked-in knowledge of CRDTs, such as using timestamps to guard data and storing the effects of conflicting operations in separate parts of the state. For example, Katara synthesizes the CRDT on the left in Figure 12 for the Add-Wins Set, which supports repeated insertions and removals by using timestamps to have Adds only shadow Removes when they are concurrent. Similarly, on the right side of Fig 12, Katara is able to discover how to use node IDs to handle non-idempotent operations in a counter CRDT, using multiple reductions in the query to take the difference of the accumulated increments and decrements.

Although our work does not focus on the runtime performance of the synthesized code, all of our CRDTs have comparable theoretical performance to human designs in existing literature. In the case of the Two-Phase Set benchmark, our synthesis algorithm comes up with a more efficient state encoding (which we discuss in Section 7.3) that simplifies the state to a single integer-to-Boolean map rather than the typical two integer sets. Overall, our synthesis algorithm is able to produce practical, provably correct CRDTs for the variety of specifications in our benchmarks.

7.2 RQ2: Search Space Pruning

A key contribution in our runtime synthesis algorithm is the use of two SMT encodings of the correctness conditions: one that can quickly verify CRDT candidates with checks for bounded executions, and one that can prove unbounded correctness but requires the synthesizer to identify additional invariants. In this section, we explore how this two-phase synthesis approach improves the performance of the overall algorithm.

First, we can compare the time that our end-to-end algorithm takes to find CRDTs for each of the benchmarks with and without the pruning optimization. In Table 2, we list the synthesis times without the bounded-log phase under the "no pruning" column. Other than lighter benchmarks which synthesize in around minute and have relatively simple CRDT state types, all the benchmarks

synthesize faster with the two-phase algorithm. The largest performance improvements come for the CRDTs with the most complex state structures: the General Counter and Add/Remove-Wins Set. In the case of the Sets, we see up to a 5x speedup by using the two-phase approach.

For a more nuanced exploration of *why* we see these speedups, we collect the time it takes each synthesis algorithm to either correctly synthesize or prune out each candidate data structure it considers for the Add-Wins Set benchmark. We compare the two algorithms in Figure 13, where we plot a distribution of the percent of candidates (out of 86 total for both) that can be evaluated within a given amount of time. With pruning, all candidate structures can be processed in under 15 minutes, allowing the end-to-end algorithm to quickly reach the state candidate that yields a verified CRDT. Without pruning, many state candidates take up to 20 minutes to be evaluated and there is a long tail of candidates that take up to an hour each. When the CRDT must have a complex state to support the specified semantics, these stragglers have a significant toll on synthesis performance since they block exploration of the program space.

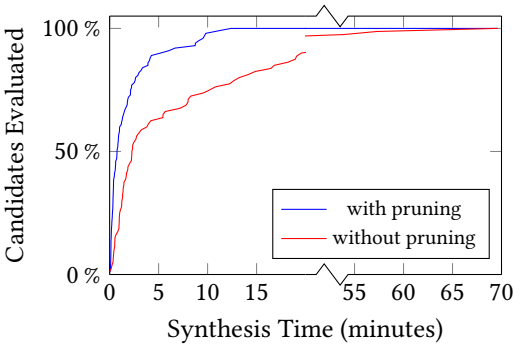

Fig. 13. The distribution of time taken to evaluate candidates for the Add-Wins Set benchmark.

7.3 RQ3: Alternative CRDT Synthesis

Finally, we evaluate the richness of the space of CRDTs our synthesis algorithm explores. In particular, we are interested in the ability of our synthesis algorithm to produce *multiple* CRDT implementations for a *single* specification. Since our combination of a sequential type and operation ordering uniquely defines the user-observable behavior of a CRDT, any alternate designs will be functionally equivalent but may have different memory utilization and performance characteristics.

In our exploration of alternate CRDT designs, we focus on the Two-Phase Set, which has moderately complex semantics since its execution has multiple phases: allow inserts, then removes, but not inserts again. When we perform synthesis, the first CRDT that is generated is surprisingly **not** the 2P-Set from existing CRDT literature, but instead (to our knowledge) a novel design that uses a map to capture the phase of each element. If we continue searching, the algorithm eventually emits the classic 2P-Set, whose state structure is larger.

We list the new design in Figure 14. There are several clever tricks that the synthesizer comes up with to match the specification while using a simpler state. First, the synthesizer realizes that the behavior of a Two-Phase Set "saturating" after removing an element matches how an `OrBool` saturates when it becomes true. Next, it discovers that by using a mapping from keys to these values, it can maintain a separate saturating value for each key in the set.

```
crdt TwoPhaseSet
  initialState: {}

  operation (m, add, value)
    return m ⊔ if add = 1
      then
        {value : false}
      else
        {value : true}

  query (m, v)
    return
      ¬m[v, default = true]
```

Fig. 14. The novel map-based CRDT that is synthesized for the Two-Phase Set benchmark.

But this still leaves a challenging situation for the initial state. Because we use the saturated *true* value to represent the element being removed, that means we have to set the value for a key

to $false$ when it is inserted the first time. But since $\bot = false$ for an `OrBool`, that would leave us with no additional state. This is where the final trick is discovered by the synthesizer: to query the map with a default value of $true$. This effectively creates a third state for when the key is not even in the map. By automatically discovering this combination of CRDT design tricks, our synthesis algorithm is able to produce this novel encoding of a Two-Phase Set.

Our synthesis algorithm also produces alternate designs for many other benchmarks, such as using pairs of clocks instead of a `LexicalProduct` for the enable/disable-wins flag benchmarks. The presence of such alternatives paves the way for future work where we synthesize not only a *correct* CRDT, but a *performant* one according to a cost model that can compare CRDT candidates. Furthermore, the pool of alternative designs may be useful for incrementally re-synthesizing CRDTs as the sequential data type is updated, something we hope to explore in the future.

8 RELATED WORK

8.1 Creating Replicated Objects from Sequential Specifications

There are a few lines of work that focus narrowly on the same problem we take aim at here: taking specifications of sequential datatypes and automatically creating equivalent replicated types. Where these projects differ from ours is primarily in our use of program synthesis: to our knowledge we are the first to utilize a search-based synthesis approach to generate state representations and runtime logic. Our other differences focus on how we chose to resolve conflicting operations, and our approach of searching semilattice compositions for the CRDT state.

Gallifrey [Milano et al. 2019], Indigo [Balegas et al. 2015], and ECROs [De Porre et al. 2021] focus on not just specifying replicated data types, but also in ensuring that applications which use them do not see inconsistent state—much as our work verifies correctness with respect to queries. Beyond the use of program synthesis in our work, the main point of divergence is in their use of preconditions and postconditions as a verification tool to exactly match the behavior of synchronous objects. Other work also uses this pre/post-condition approach [De Porre et al. 2019] or shares the goal of matching sequential behavior [Soethout et al. 2019]. Our goal is not to exactly match sequential behavior; we let programmers tweak semantics with ordering constraints on conflicting operations. This allows us to lift all specified operations into CRDTs, rather than limiting ourselves to operations that already commute (as in Gallifrey) or resorting to explicit synchronization or deferred re-execution for conflicting events (as in Indigo and ECROs).

The MRDT line of work [Kaki et al. 2019; Soundarapandian et al. 2022] starts with a similar premise to ours—creating replicated datatypes from annotated sequential specifications—but takes a radically different approach. Our largest differences center around their runtime system, which is based on a Git-inspired log of versioned data structures, and in their mode of annotation, which centers around abstraction and concretization functions. In contrast, our sorting-based annotations are simpler for non-experts to reason about, and our generated CRDTs require only a standard gossip protocol. Additionally, the merge function of MRDTs are generated using a rule-based approach, whereas our work takes the search-based synthesis approach.

8.2 Program Synthesis and Verified Lifting

The synthesis approach taken in this work is directly inspired by verified lifting [Kamil et al. 2016], the approach at the heart of work such as Domino [Sivaraman et al. 2016], Casper [Ahmad and Cheung 2018], and Dexter [Ahmad et al. 2019]. With verified lifting, the correctness conditions for synthesizing code in a particular DSL are derived from *existing implementations* in standard languages such as C/C++. Our approach expands on this tradition primarily by our choice to synthesize entire data types, instead of just function implementations. This involves more complex

verification conditions that check *behavioral equivalence* between the input code and the synthesized CRDT, rather than just checking equality of function outputs, and requires synthesizing more complex invariants to enable verification of unbounded interactions. Past work has explored the formal foundations of specifying CRDT correctness in terms of a sequential reference by layering constraints on how the effects of operations are applied [Burckhardt et al. 2014]. Our introduction of pairwise ordering constraints, guided by this work, enables fully automated verification of CRDTs with lightweight annotations that are easy for non-experts to write.

Few previous systems have attempted to directly apply search-based program synthesis to the space of replication. Two that stand out are Hamsaz [Houshmand and Lesani 2019] and Hampa [Li et al. 2020]. Hamsaz uses programmer-provided invariants to synthesize custom consistency protocols for the replication of shared data structures. While its analysis component is reminiscent of other work, such as Quelea, and the Indigo line [Balegas et al. 2018, 2015; De Porre et al. 2021; Sivaramakrishnan et al. 2015], its novel synthesis component is of particular interest to this work. Like our work, Hamsaz uses an SMT encoding of the programmer-specified semantics to search through potential replication strategies. Hampa [Li et al. 2020], a similar work from the same research group, adds recency to the mix. However, both of these solutions are focused on identifying efficient coordination protocols, rather than *eliminating* coordination altogether. The CRDTs synthesized by our algorithm can be replicated without needing any coordination.

8.3 Verifying Replicated Data Types

Many previous systems have also provided verification systems for manually-implemented CRDTs, checking both convergence properties and correctness with respect to a specification. Several of these require manual proof effort [Gomes et al. 2017; Liu et al. 2020; Zeller et al. 2014], which make them infeasible for program synthesis approaches that require rapid verification of a large number of candidates without any human involvement.

Of particular note is recent work that explores automated verification of convergence properties via an SMT encoding [Nagar and Jagannathan 2019], which is especially relevant because we also use SMT to automatically verify CRDT candidates. However, our work differs from this research in *what* is being verified. Because our CRDTs are *convergent by construction*, we do not need to perform any convergence verification. Instead, our verification conditions focus on the *user-observable behavior* of the CRDT, including checking the correctness of queries—something that convergence verification approaches do not include (as queries do not affect convergence).

Other lines of work focus on the general question of correct use of weak consistency [Gotsman et al. 2016; Wang et al. 2019], which is a wider problem that is not specific to CRDTs. Furthermore, these lines of work focus on reasoning about how application invariants can be maintained when using weak consistency under the hood, rather than how different types of state can be replicated in an eventually consistent manner. Indeed, the literature around safely using weak consistency is complementary to our contributions, as they provide a path for developers to safely build applications on top of the CRDTs we synthesize.

There are certain CRDTs that challenge the current limitations of what we can specify in SMT for verification. These make for interesting future research directions. One example is the floating point comparison used in Logoot [Weiss et al. 2009] which would require richer user-specified orderings and extensions to the query language. Another example is the Replicated Growable Array (RGA), which requires tracking sequential data. Our query language does not currently support the types of iterative computations required to reason about sequences, but it has been shown that RGA can be represented as Datalog queries [Kleppmann 2018] over operations. We see this Datalog representation as a promising direction for enabling synthesis of such advanced CRDTs.

8.4 Making Replicated Objects Easier to Work With

Several papers attempt to make the process of programming against weak consistency tractable. Some do so by exposing weakly-consistent replicated objects with approachable semantics; indeed, the original CRDT work fits in this vein [Shapiro et al. 2011a]. Other such approaches include the CloudTypes work from Microsoft [Burckhardt et al. 2012] or work on allowing an application to safely mix consistency levels via MixT [Milano and Myers 2018], Disciplined Inconsistency [Holt et al. 2016], CScript [De Porre et al. 2020], or Red-Blue consistency [Li et al. 2012]. Some work automatically chooses consistency levels for the programmer, driving the choice of mixed consistency via program invariants rather than explicit consistency annotations [Kaki et al. 2018; Li et al. 2014; Sivaramakrishnan et al. 2015; Zhao and Haller 2018, 2020]. All work on mixing consistency shares the belief that programmers require stronger consistency for some operations; in contrast, we let users introduce semantic adjustments that eliminate the need for strong consistency.

Other work focuses on ensuring that application executions are convergent despite the inherent non-determinism of concurrency and weakly-consistent replication. These include a long line of work from Berkeley [Alvaro et al. 2014, 2011, 2010; Conway et al. 2014, 2012], and the Gallifrey, LASP, and LVars languages [Kuper and Newton 2013; Meiklejohn and Van Roy 2015; Milano et al. 2019]. While we believe that whole-language approaches are valuable in this space, the CRDT literature typically does not analyze programs beyond the boundaries of the CRDT implementation.

9 CONCLUSION

The future of computing is distributed, so it is important to reduce the complexity of developing correct, efficient distributed programs. We believe that verified lifting can be a useful tool towards this goal, by automating much of the process of converting familiar sequential logic into scalable distributed code. In this paper, we presented a first step in this direction with Katara, a system that automatically synthesizes CRDT designs from existing sequential data type implementations, requiring only simple annotations that are are easy for developers to reason about.

We formalized the definition of *correctness* for CRDTs in terms of a sequential data type by introducing *operation orderings*, which allow users to define how the CRDT should handle conflicting non-commutative operations. To automate the verification process, we developed SMT encodings of this correctness definition that can be used to check CRDT candidates with a solver. Finally, we explored how compositions of semilattices can be naturally used as the state of a CRDT, and defined grammars for runtime logic and the invariants that enable unbounded verification of correctness. Our end-to-end algorithm efficiently synthesizes CRDTs for a variety of scenarios and produces novel alternatives to human-designed CRDTs. With Katara, we hope to further unlock the power of distributed systems by making it possible for any developer to automatically replicate their existing data types by synthesizing provably equivalent CRDTs.

ACKNOWLEDGMENTS

We thank Audrey Cheng, David Chu, Natacha Crooks, and our anonymous reviewers for their insightful feedback on this paper. This work is supported in part by National Science Foundation CISE Expeditions Award CCF-1730628, IIS-1955488, IIS-2027575, DOE award DE-SC0016260, ARO award W911NF2110339, and ONR award N00014-21-1-2724, and by gifts from Amazon Web Services, Ant Group, Ericsson, Futurewei, Google, Intel, Meta, Microsoft, Scotiabank, and VMware. Shadaj Laddad is supported in part by the NSF Graduate Research Fellowship Program under Grant No. DGE 2146752. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Maaz Bin Safeer Ahmad and Alvin Cheung. 2018. Automatically Leveraging MapReduce Frameworks for Data-Intensive Applications. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1205–1220. https://doi.org/10.1145/3183713.3196891

Maaz Bin Safeer Ahmad, Jonathan Ragan-Kelley, Alvin Cheung, and Shoaib Kamil. 2019. Automatically Translating Image Processing Libraries to Halide. *ACM Trans. Graph.* 38, 6, Article 204 (nov 2019), 13 pages. https://doi.org/10.1145/3355089.3356549

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. 1–8. https://doi.org/10.1109/FMCAD.2013.6679385

Peter Alvaro, Neil Conway, Joseph M Hellerstein, and David Maier. 2014. Blazes: Coordination analysis for distributed programs. In *2014 IEEE 30th International Conference on Data Engineering*. IEEE, 52–63.

Peter Alvaro, Neil Conway, Joseph M Hellerstein, and William R Marczak. 2011. Consistency Analysis in Bloom: a CALM and Collected Approach.. In *CIDR*. Citeseer, 249–260.

Peter Alvaro, William R Marczak, Neil Conway, Joseph M Hellerstein, David Maier, and Russell Sears. 2010. Dedalus: Datalog in time and space. In *International Datalog 2.0 Workshop*. Springer, 262–281.

Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, and Nuno Preguiça. 2018. IPA: Invariant-Preserving Applications for Weakly Consistent Replicated Databases. *Proc. VLDB Endow.* 12, 4 (dec 2018), 404–418. https://doi.org/10.14778/3297753.3297760

Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. 2015. Putting Consistency Back into Eventual Consistency. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 6, 16 pages. https://doi.org/10.1145/2741948.2741972

Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13243)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. 2012. Cloud Types for Eventual Consistency. In *ECOOP 2012 – Object-Oriented Programming*, James Noble (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 283–307.

Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated data types: specification, verification, optimality. *ACM Sigplan Notices* 49, 1 (2014), 271–284.

Alvin Cheung, Natacha Crooks, Joseph M Hellerstein, and Mae Milano. 2021. New directions in cloud programming. *The Conference on Innovative Data Systems Research (CIDR)*, 14 pages.

Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing database-backed applications with query synthesis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 3–14.

Neil Conway, Peter Alvaro, Emily Andrews, and Joseph M Hellerstein. 2014. Edelweiss: Automatic storage reclamation for distributed programming. *Proceedings of the VLDB Endowment* 7, 6 (2014), 481–492.

Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. 2012. Logic and Lattices for Distributed Programming. In *Proceedings of the Third ACM Symposium on Cloud Computing* (San Jose, California) *(SoCC '12)*. Association for Computing Machinery, New York, NY, USA, Article 1, 14 pages. https://doi.org/10.1145/2391229.2391230

Natacha Crooks, Youer Pu, Nancy Estrada, Trinabh Gupta, Lorenzo Alvisi, and Allen Clement. 2016. TARDiS: A Branch-and-Merge Approach To Weak Consistency. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1615–1628. https://doi.org/10.1145/2882903.2882951

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

Kevin De Porre, Carla Ferreira, Nuno Preguiça, and Elisa Gonzalez Boix. 2021. ECROs: Building Global Scale Systems from Sequential Code. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 107 (oct 2021), 30 pages. https://doi.org/10.1145/3485484

Kevin De Porre, Florian Myter, Christophe De Troyer, Christophe Scholliers, Wolfgang De Meuter, and Elisa Gonzalez Boix. 2019. Putting Order in Strong Eventual Consistency. In *Distributed Applications and Interoperable Systems*, José Pereira

and Laura Ricci (Eds.). Springer International Publishing, Cham, 36–56.

Kevin De Porre, Florian Myter, Christophe Scholliers, and Elisa Gonzalez Boix. 2020. CScript: A distributed programming language for building mixed-consistency applications. *J. Parallel and Distrib. Comput.* 144 (2020), 109–123. https://doi.org/10.1016/j.jpdc.2020.05.010

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review* 41, 6 (2007), 205–220.

Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. 1987. Epidemic Algorithms for Replicated Database Maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing* (Vancouver, British Columbia, Canada) *(PODC '87)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/41840.41841

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. https://doi.org/10.1145/3133933

Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2016. 'Cause I'm Strong Enough: Reasoning about Consistency Choices in Distributed Systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 371–384. https://doi.org/10.1145/2837614.2837625

Joseph M Hellerstein and Peter Alvaro. 2020. Keeping CALM: when distributed consistency is easy. *Commun. ACM* 63, 9 (2020), 72–81.

Todd Hoff. 2014. *How League Of Legends Scaled Chat To 70 Million Players - It Takes Lots Of Minions.* http://highscalability.com/blog/2014/10/13/how-league-of-legends-scaled-chat-to-70-million-players-it-t.html

Brandon Holt, James Bornholt, Irene Zhang, Dan Ports, Mark Oskin, and Luis Ceze. 2016. Disciplined Inconsistency with Consistency Types. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (Santa Clara, CA, USA) *(SoCC '16)*. Association for Computing Machinery, New York, NY, USA, 279–293. https://doi.org/10.1145/2987550.2987559

Farzin Houshmand and Mohsen Lesani. 2019. Hamsaz: Replication Coordination Analysis and Synthesis. *Proc. ACM Program. Lang.* 3, POPL, Article 74 (jan 2019), 32 pages. https://doi.org/10.1145/3290387

Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Boston, MA) *(USENIXATC'10)*. USENIX Association, USA, 11.

Gowtham Kaki, Kapil Earanky, KC Sivaramakrishnan, and Suresh Jagannathan. 2018. Safe Replication through Bounded Concurrency Verification. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 164 (oct 2018), 27 pages. https://doi.org/10.1145/3276534

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. https://doi.org/10.1145/3360580

Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Verified Lifting of Stencil Computations. *SIGPLAN Not.* 51, 6 (jun 2016), 711–726. https://doi.org/10.1145/2980983.2908117

Martin Kleppmann. 2018. Data structures as queries: Expressing CRDTs using Datalog. (2018). https://martin.kleppmann.com/2018/02/26/dagstuhl-data-consistency.html

Martin Kleppmann. 2022. *Assessing the understandability of a distributed algorithm by tweeting buggy pseudocode.* Technical Report. University of Cambridge, Computer Laboratory.

Martin Kleppmann and Alastair R. Beresford. 2017. A Conflict-Free Replicated JSON Datatype. *IEEE Transactions on Parallel and Distributed Systems* 28, 10 (2017), 2733–2746. https://doi.org/10.1109/TPDS.2017.2697382

Rusty Klophaus. 2010. Riak Core: Building Distributed Applications without Shared State. In *ACM SIGPLAN Commercial Users of Functional Programming* (Baltimore, Maryland) *(CUFP '10)*. Association for Computing Machinery, New York, NY, USA, Article 14, 1 pages. https://doi.org/10.1145/1900160.1900176

Lindsey Kuper and Ryan R. Newton. 2013. LVars: Lattice-Based Data Structures for Deterministic Parallelism. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-Performance Computing* (Boston, Massachusetts, USA) *(FHPC '13)*. Association for Computing Machinery, New York, NY, USA, 71–84. https://doi.org/10.1145/2502323.2502326

Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (apr 2010), 35–40. https://doi.org/10.1145/1773912.1773922

Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (jul 1978), 558–565. https://doi.org/10.1145/359545.359563

Leslie Lamport. 1998. The part-time parliament. *ACM Transactions on Computer Systems* 16, 2 (May 1998), 133–169. https://doi.org/10.1145/279227.279229

Cheng Li, Joao Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. 2014. Automating the Choice of Consistency Levels in Replicated Systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX

Association, Philadelphia, PA, 281–292. https://www.usenix.org/conference/atc14/technical-sessions/presentation/li_cheng_2

Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, 265–278. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/li

Xiao Li, Farzin Houshmand, and Mohsen Lesani. 2020. Hampa: Solver-Aided Recency-Aware Replication. In *Computer Aided Verification*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 324–349.

Yiyun Liu, James Parker, Patrick Redmond, Lindsey Kuper, Michael Hicks, and Niki Vazou. 2020. Verifying Replicated Data Types with Typeclass Refinements in Liquid Haskell. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 216 (nov 2020), 30 pages. https://doi.org/10.1145/3428284

Christopher Meiklejohn and Peter Van Roy. 2015. Lasp: A Language for Distributed, Coordination-Free Programming. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming* (Siena, Italy) *(PPDP '15)*. Association for Computing Machinery, New York, NY, USA, 184–195. https://doi.org/10.1145/2790449.2790525

Mae Milano and Andrew C. Myers. 2018. MixT: A Language for Mixing Consistency in Geodistributed Transactions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 226–241. https://doi.org/10.1145/3192366.3192375

Matthew Milano, Rolph Recto, Tom Magrino, and Andrew C. Myers. 2019. A Tour of Gallifrey, a Language for Geodistributed Programming. In *3rd Summit on Advances in Programming Languages (SNAPL 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 136)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 11:1–11:19. https://doi.org/10.4230/LIPIcs.SNAPL.2019.11

Kartik Nagar and Suresh Jagannathan. 2019. Automated Parameterized Verification of CRDTs. https://doi.org/10.48550/ARXIV.1905.05684

Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Philadelphia, PA) *(USENIX ATC'14)*. USENIX Association, USA, 305–320.

Raymond Roestenburg, Rob Williams, and Robertus Bakker. 2016. *Akka in action*. Simon and Schuster.

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011a. *A comprehensive study of convergent and commutative replicated data types*. Ph.D. Dissertation. Inria–Centre Paris-Rocquencourt; INRIA.

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011b. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*. Springer, 386–400.

KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. 2015. Declarative Programming over Eventually Consistent Data Stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 413–424. https://doi.org/10.1145/2737924.2737981

Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet Transactions: High-Level Programming for Line-Rate Switches. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 15–28. https://doi.org/10.1145/2934872.2934900

Tim Soethout, Tijs van der Storm, and Jurgen J. Vinju. 2019. Static Local Coordination Avoidance for Distributed Objects. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control* (Athens, Greece) *(AGERE 2019)*. Association for Computing Machinery, New York, NY, USA, 21–30. https://doi.org/10.1145/3358499.3361222

Vimala Soundarapandian, Adharsh Kamath, Kartik Nagar, and KC Sivaramakrishnan. 2022. Certified Mergeable Replicated Data Types. In *Proceedings of the 43rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 16 pages.

Emina Torlak and Rastislav Bodik. 2013. Growing Solver-Aided Languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Indianapolis, Indiana, USA) *(Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 135–152. https://doi.org/10.1145/2509578.2509586

Werner Vogels. 2009. Eventually Consistent. *Commun. ACM* 52, 1 (jan 2009), 40–44. https://doi.org/10.1145/1435417.1435432

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. https://doi.org/10.1145/3314221.3314617

Stephane Weiss, Pascal Urso, and Pascal Molli. 2009. Logoot: A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks. In *2009 29th IEEE International Conference on Distributed Computing Systems*. 404–412. https://doi.org/10.1109/ICDCS.2009.75

Chenggang Wu, Jose Faleiro, Yihan Lin, and Joseph Hellerstein. 2018. Anna: A KVS for Any Scale. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 401–412. https://doi.org/10.1109/ICDE.2018.00044

Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. 2014. Formal Specification and Verification of CRDTs. In *Formal Techniques for Distributed Objects, Components, and Systems*, Erika Ábrahám and Catuscia Palamidessi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 33–48.

Xin Zhao and Philipp Haller. 2018. Observable Atomic Consistency for CvRDTs. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control* (Boston, MA, USA) *(AGERE 2018)*. Association for Computing Machinery, New York, NY, USA, 23–32. https://doi.org/10.1145/3281366.3281372

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. https://doi.org/10.1016/j.jlamp.2020.100561