ECROs: Building Global Scale Systems from Sequential Code

KEVIN DE PORRE, Vrije Universiteit Brussel, Belgium
CARLA FERREIRA, NOVA School of Science and Technology, Portugal
NUNO PREGUIÇA, NOVA School of Science and Technology, Portugal
ELISA GONZALEZ BOIX, Vrije Universiteit Brussel, Belgium

To ease the development of geo-distributed applications, replicated data types (RDTs) offer a familiar programming interface while ensuring state convergence, low latency, and high availability. However, RDTs are still designed exclusively by experts using ad-hoc solutions that are error-prone and result in brittle systems. Recent works statically detect conflicting operations on existing data types and coordinate those at runtime to guarantee convergence and preserve application invariants. However, these approaches are too conservative, imposing coordination on a large number of operations. In this work, we propose a principled approach to design and implement efficient RDTs taking into account application invariants. Developers extend sequential data types with a distributed specification, which together form an RDT. We statically analyze the specification to detect conflicts and unravel their cause. This information is then used at runtime to serialize concurrent operations safely and efficiently. Our approach derives a correct RDT from any sequential data type without changes to the data type's implementation and with minimal coordination. We implement our approach in Scala and develop an extensive portfolio of RDTs. The evaluation shows that our approach provides performance similar to conflict-free replicated data types for commutative operations, and considerably improves the performance of non-commutative operations, compared to existing solutions.

CCS Concepts: • **Computing methodologies** → *Distributed computing methodologies*; • **Software and its engineering** → *Automated static analysis*; • **Theory of computation** → *Distributed algorithms*; • **Computer systems organization** → *Availability*; *Cloud computing*.

Additional Key Words and Phrases: replication, data structures, eventual consistency

ACM Reference Format:
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 (October 2021), 30 pages. https://doi.org/10.1145/3485484

1 INTRODUCTION

Geo-replication is a popular technique employed by distributed applications to reduce user-observed latencies as replicas are geographically closer to the clients. Developers of geo-distributed applications, however, face a difficult choice between availability and consistency [Brewer 2012, 2000; Kleppmann 2015]. Ensuring strong consistency requires coordination to enforce a total order of updates across all replicas. This increases latency, which translates into reduced performance, and decreased (offline) availability. When adopting weaker consistency guarantees (e.g. eventual consistency (EC) [Vogels 2009]), replicas can execute operations without coordination. Operations

Authors' addresses: Kevin De Porre, kevin.de.porre@vub.be, Vrije Universiteit Brussel, Pleinlaan 2, Brussels, Belgium, 1050; Carla Ferreira, NOVA School of Science and Technology, Caparica, Portugal, carla.ferreira@fct.unl.pt; Nuno Preguiça, NOVA School of Science and Technology, Caparica, Portugal, nuno.preguica@fct.unl.pt; Elisa Gonzalez Boix, egonzale@vub.be, Vrije Universiteit Brussel, Pleinlaan 2, Brussels, Belgium, 1050.

2475-1421/2021/10-ART107
https://doi.org/10.1145/3485484

are propagated in the background, leading to different execution orders at different replicas. This improves performance at the cost of correctness, as it may introduce (temporary) state divergence and violate application invariants [Bailis et al. 2014; Brewer 2000].

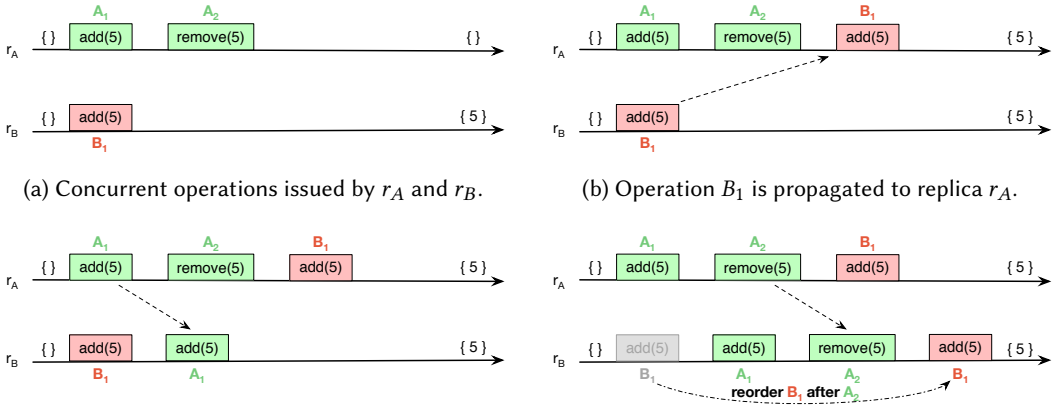
To ease the development of geo-distributed applications, much work has studied the concept of a replicated data type (RDT) [Burckhardt et al. 2012; De Porre et al. 2019; Kaki et al. 2019; Shapiro et al. 2011b]. The goal of an RDT is to expose the same interface as its sequential counterpart and embed in its implementation mechanisms to enforce correctness. Unfortunately, designing new RDTs that (1) guarantee state convergence and (2) preserve application invariants is currently cumbersome and reserved to experts. Although there has been a lot of active research with respect to RDTs [Almeida et al. 2015; Baquero et al. 2017; Burckhardt et al. 2012; De Porre et al. 2019; Kermarrec et al. 2001; Shapiro et al. 2011b; Terry et al. 1995], there is no principled approach to turn existing data types into efficient RDTs [Baquero et al. 2017; Kaki et al. 2018; Weidner et al. 2020].

Much work [Almeida et al. 2015; Baquero et al. 2017; Shapiro et al. 2011b] focuses primarily on state convergence, relying on mathematical properties to guarantee convergence by design, e.g. conflict-free replicated data types (CRDTs) rely on commutative operations [Shapiro et al. 2011b]. These approaches tend to be limited in scope (e.g. to a specific data type), and designing new RDTs is hard, as the implementation must obey those mathematical properties [Weidner et al. 2020]. Bayou [Terry et al. 1995] and Cloud Types [Burckhardt et al. 2012] require programmers to devise custom merge procedures that deal with conflicts. However, devising correct merge procedures is a difficult task. Mergeable replicated data types [Kaki et al. 2019] automatically derive merge procedures from invertible relational specifications that convert the RDT's state to relations over sets and back. However, the merge semantics can be hard to customize as they depend on the set relations being used. De Porre et al. [2019]; Kermarrec et al. [2001] totally order sequential operations based on semantic information. But finding such a serialization at runtime induces high latencies and yields poor scalability.

With respect to invariant preservation, much work statically analyzes data types to detect operations that violate application-level invariants. Balegas et al. [2018, 2015]; Gotsman et al. [2016]; Kaki et al. [2018]; Li et al. [2012, 2018]; Sivaramakrishnan et al. [2015] start from an existing RDT and extend them with invariants that are enforced by coordinating unsafe operations, or by resorting to a stronger consistency model. However, those approaches do not aid with building correct RDTs which is known to be difficult and error-prone [Baquero et al. 2017; Kaki et al. 2018; Kleppmann and Beresford 2017]. Hamsaz [Houshmand and Lesani 2019] and Hampa [Li et al. 2020] derive coordination protocols from a data type's specification. Overall, the aforementioned approaches tend to be conservative, leading to unnecessary coordination (cf. Section 2.3). This impacts the latency observed by users and reduces the scalability and availability of the system.

The ECRO Way. Our work rethinks the concept of a replicated data type to provide a principled approach towards the design and implementation of efficient RDTs with respect for application invariants. An RDT consists of a sequential data type and a distributed specification describing the semantics of its operations. The resulting RDT adheres to the provided specification with minimal coordination. This simplifies the adoption of RDTs as it is possible to convert *any* sequential data type into an RDT without having to reason about conflicts and without restrictions on the operations.

We introduce Explicitly Consistent Replicated Objects (ECROs): RDTs that are derived from sequential data types, based on a distributed specification that declares the application semantics by means of invariants over replicated state. ECROs statically detect conflicting operations and identify solutions beforehand. This information is used at runtime to avoid conflicts by locally (re-)ordering conflicting operations when possible, and coordinating operations only if correctness



(a) Concurrent operations issued by r_A and r_B.

(b) Operation B_1 is propagated to replica r_A.

(c) Operation A_1 is propagated to replica r_B.

(d) Operation A_2 is propagated to replica r_B which reorders operation B_1 to ensure add-wins semantics.

Fig. 1. Reordering operations in a replicated Add-Wins Set ECRO.

cannot be guaranteed otherwise. For example, `add(x)` and `remove(y)` operations on a replicated set do not commute when $x = y$. To ensure convergence, CRDTs propose special (ad-hoc) set implementations that make the operations commutative. Other approaches such as Hamsaz would coordinate `adds` and `removes`. In contrast, ECROs locally reorder concurrent `adds` and `removes` such that replicas execute them in the *same* order. Note that add-wins set semantics can be achieved by first applying `remove` operations and only then applying concurrent `add` operations. ECROs statically derive this ordering information from the set's invariants. Figure 1 shows the example of an Add-Wins Set ECRO which initially is empty {}. In Fig. 1a, replica r_A adds 5 to the set (operation A_1) and later removes it from the set (operation A_2). Concurrently, replica r_B also adds 5 to the set (operation B_1). Operation B_1 is concurrent with operations A_1 and A_2. Then in Fig. 1b, operation B_1 is propagated to replica r_A which adds 5 to the set. Similarly, in Fig. 1c, operation A_1 is propagated to replica r_B which adds 5 to the set. This operation leaves the state unchanged since 5 was already in the set. When operation A_2 is propagated to replica r_B (cf. Fig. 1d), r_B cannot immediately apply A_2 since removing 5 from the set would violate the desired add-wins semantics. Instead, r_B *reorders* the concurrent `add(5)` (operation B_1) such that it is executed after `remove(5)` (operation A_2) and thus guarantees add-wins semantics.

The ECRO approach radically differs from existing approaches in three ways. First, correct RDTs are derived from sequential data types and their accompanying specification. By decoupling the data type's implementation from its distributed semantics we improve the maintainability of RDTs. For example, one can change the semantics of a replicated set (e.g. add-wins, remove-wins, or last-writer-wins set) by attaching a different specification to the set. In contrast, related work requires different, non-trivial implementations as described by Shapiro et al. [2011a]. Second, ECROs totally order concurrent operations if they do not commute and thus guarantees convergence without coordination. Similarly, unsafe operations may run concurrently if a safe reordering exists. In contrast, existing approaches [Balegas et al. 2015; Houshmand and Lesani 2019; Li et al. 2012, 2018, 2020] impose coordination on all non-commutative operations and unsafe operations. Finally, ECROs ignore causality between unrelated commutative operations while prior works always execute operations in an order that is compatible with causal order. This increases the flexibility in the execution of operations, while still enforcing relevant causal dependencies, allowing ECROs to

implement semantics that cannot be achieved in other systems that totally order operations. For example, `add(5)` and `remove(6)` operations are unrelated since they affect different elements. The causal relation between those operations is thus irrelevant and we can execute these operations in any order.

In summary, the contributions of this paper are:

- An algorithm to derive ECROs from a sequential data type and its distributed specification, and proofs of state convergence [Shapiro et al. 2011b] and safety.
- Ordana, a static analysis tool combining a novel safety analysis with existing dependency [Houshmand and Lesani 2019] and commutativity analyses [Balegas et al. 2015] to detect and solve conflicts using a novel approach that reorders conflicting operations locally in order to guarantee convergence and preserve invariants without coordination.
- A full-fledged Scala implementation of our approach that includes an extensive portfolio of RDTs, covering a wide variety of existing RDTs (sets, maps, lists, etc.) as well as new RDTs. We also build a replicated auction system from sequential data types only, and without relying on ad-hoc solutions.
- A performance evaluation which shows that ECROs provide latency similar to CRDTs for commutative operations, and significantly reduce the latency of non-commutative operations and unsafe operations when compared to RedBlue [Li et al. 2012] and PoR [Li et al. 2018].

Availability. The complete portfolio of RDTs implemented with ECROs and the scripts to reproduce the performance evaluation are available in our software artifact at https://doi.org/10.5281/zenodo.5510036.

2 BUILDING GEO-DISTRIBUTED APPLICATIONS, THE ECRO WAY

We now present our novel approach to programming cloud computing applications running on geo-distributed data centers by means of Explicitly Consistent Replicated Objects (ECROs). We provide an overview of the ECRO approach, and show how it can be applied to implement replicated sets and a replicated auction system, and discuss how it differs from state-of-the-art approaches.

2.1 Overview

Figure 2 depicts a high-level overview of the ECRO approach. To build an RDT, programmers extend *sequential* data types with a *distributed specification* defining the data type's semantics by means of invariants over replicated state. Together, the sequential data type and its distributed specification form an ECRO. The state of an ECRO is replicated across machines, each of which is

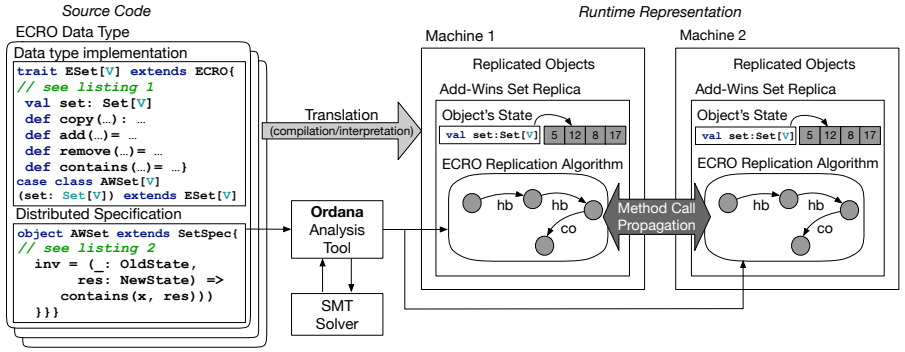


Fig. 2. Overview of ECROs.

```
trait ESet[V] extends ECRO {
  val set: Set[V]
  def copy(set: Set[V]): ESet[V]
  def add(x: V) = copy(set + x)
  def remove(x: V) = copy(set - x)
  def contains(x: V) = set.contains(x) }
```

Listing 1. Sequential set implementation.

```
case class AWSet[V](set: Set[V]) extends ESet[V]
case class RWSet[V](set: Set[V]) extends ESet[V]
object AWSet {
  val contains: Relation = ... // see Appendix A.1 in [De Porre et al. 2021]
  postcondition of add {
    (old: OldState, res: NewState) =>
      contains(x, res) /\ contains.copyExcept(old -> res, elem === x) }
  postcondition of remove {
    (old: OldState, res: NewState) =>
      not (contains(x, res)) /\ contains.copyExcept(old -> res, elem === x) }
  invariant on add {
    (_: OldState, res: NewState) => contains(x, res)) } }
object RWSet {
  // contains & postconditions same as AWSet
  invariant on remove {
    (_: OldState, res: NewState) => not (contains(x, res)))) } }
```

Listing 2. Add-Wins and Remove-Wins Set ECROs.

said to hold a *replica*. Programmers interact with ECROs by calling *methods* on a replica. Method calls are propagated between replicas using a broadcasting mechanism that guarantees at-least-once causal delivery. Under these assumptions, the ECRO replication algorithm guarantees safety and strong convergence. *Safety* is the property that the replicated state respects the application's invariants. *Strong convergence* [Shapiro et al. 2011b] is the property that correct replicas that processed the same calls (possibly in a different order) are in equivalent states. Key to guaranteeing these properties is our analysis tool, called Ordana, that statically analyzes distributed specifications to detect conflicting operations and find solutions beforehand. At runtime, the ECRO replicas use the information inferred by Ordana to serialize calls efficiently (i.e. with minimal coordination) while upholding safety and strong convergence. To this end, every replica keeps a tentative serialization of the calls which may be affected by concurrent calls. When calls stabilize across all replicas, they are committed, i.e. they are applied in order on the state before being garbage collected.

2.2 Building Replicated Sets

We now illustrate our approach by means of a set RDT which can be found in many geo-distributed applications. A set RDT differs from a sequential set as mutiple users may add and remove the same element concurrently. When these updates have been received by all replicas, the element must be present in all replicas (add-wins semantics) or absent from all replicas (remove-wins semantics). Hence, a sequential data type may have several replicated counterparts, each exhibiting different semantics when facing concurrent operations.

ECROs let programmers turn any sequential data type into an RDT by defining the desired semantics in the data type's distributed specification. The specification describes the operations that modify the internal state by means of four components: a context, a precondition, a postcondition, and an invariant. Each component (cf. Section 3.1) is a function that is parametrized by the data type's state(s) and returns a first-order logic formula used by Ordana to analyze the operations. Listing 2 shows part of the implementation of the add-wins `AWSet` and remove-wins `RWSet` ECROs in Scala[1]. Both sets extend the `ESet` trait (shown in Listing 1) which wraps Scala's built-in immutable set and offers the typical set operations. The sets' distributed specifications use an embedded domain specific language (DSL) that we built for programming with first-order logic (cf. Appendix A in [De Porre et al. 2021]). By convention, the specification is defined in the class' companion object. The postconditions for `add(x)` and `remove(x)` state that after adding/removing `x`, the element is present/absent from the resulting state `res`, and that all other elements are unchanged (lines 7 and 10). The `AWSet` contains an invariant on the `add(x)` operation to force element `x` to be present in the resulting state `res` and thus guarantees add-wins semantics (lines 11-12). Similarly, the `RWSet` contains an invariant on the `remove(x)` operation to force element `x` to be absent from the resulting state and thus guarantees remove-wins semantics (lines 15-16).

This example demonstrates the flexibility of ECROs: to switch between add-wins and remove-wins semantics, *only* the invariant defined in the distributed specification was changed (one line of code). In contrast, state-of-the-art RDT solutions like CRDTs require two different data type implementations each engineered to yield the desired semantics (cf. Section 6.2.1). Other works, like RedBlue consistency, do not require changes to the data type but would synchronize all `add` and `remove` operations due to the possibility of an add-remove conflict. As we show in Section 4, Ordana finds a solution to add-remove conflicts that does not require coordination.

2.3 Building a Geo-Distributed Auction System

We now show how to use the ECRO approach to build a custom RDT for which no ready-made RDT exists. To this end, we develop a geo-distributed eBay-like auction system akin to the RUBiS system [Cecchet and Marguerite 2009], where users open auctions, bid on auctions, and close auctions. RUBiS requires usernames to be unique and each bid must be linked to one existing user.

In an attempt to develop RUBiS, one may compose a set RDT of users with a map RDT from auction IDs to auctions where an auction consists of a set RDT of bids and an enable-once flag RDT indicating whether the auction is open or closed. However, state-of-the-art RDT solutions (CRDTs, Cloud Types, etc.) require programmers to manually uphold application invariants. For example, two users may concurrently register under the same username, violating one of RUBiS' invariants. It is also not clear how to ensure that each bid is linked to one existing user (i.e. referential integrity) as this would require an atomic update across RDTs.

We now discuss how to implement a replicated RUBiS system starting from a sequential implementation, using ECROs[2]. The sequential implementation keeps a set of users and a map from auction IDs to auctions containing a set of bids, a status (open or closed), and a winner. When a user bids on an auction, it is added to the set of bids. Bids may only be placed on open auctions. Listing 3 shows the distributed specification of the `placeBid` and `closeAuction` methods. The precondition of `placeBid` (line 8 to 10) requires the auction to be open, the user to exist, and the bid to be bigger than zero. The postcondition of `placeBid` (line 11 to 14) extends the state with the new bid and copies all bids from the old state to the new state. The postcondition of `closeAuction`

[1]The syntax is simplified for presentation purposes. The complete implementation of the sets is provided in Appendix A.1 in [De Porre et al. 2021].

[2]The complete implementation of RUBiS is provided in Appendix A.2 in [De Porre et al. 2021].

```
1  case class Rubis(users: Set[User], auctions: Map[AID, Auction]) extends ECRO {
2    def placeBid(auctionId: AID, userId: User, price: Int): Rubis = ...
3    def closeAuction(auctionId: AID): Rubis = ...
4  }
5  object Rubis {
6    // for the definition of the relations, see Appendix A.2 in [De Porre et al. 2021]
7    val auction: Relation = ...; val user: Relation = ...; val bid: Relation = ...
8    precondition of placeBid {
9      (state: CurrentState) =>
10       auction(auctionId, Open, state) /\ user(userId, state) /\ (price >> 0) }
11   postcondition of placeBid {
12     (old: OldState, res: NewState) =>
13       old + bid(auctionId, userId, price, res) /\ bid.copy(old -> res)
14   }
15   postcondition of closeAuction {
16     (old: OldState, res: NewState) =>
17       old + auction(auctionId, Closed, res) /\ not(auction(auctionId, Open, res)) /\
18       auction.copyExcept(old -> res, id === auctionId)
19   }
20 }
```

Listing 3. Distributed specification of an auction system.

(line 15 to 19) puts the auction's status on closed, states that it can no longer be open, and copies all other auctions from the old state to the new one.

By statically analyzing the distributed specification, Ordana detects operations that may violate application invariants. For example, using the precondition of `placeBid`, Ordana detects that concurrent `placeBid` and `closeAuction` calls may lead to a bid being placed on a closed auction if a replica applies `closeAuction` before `placeBid`. ECROs solve this conflict by imposing an ordering on the operations (cf. Section 5). In contrast, existing approaches (RedBlue, PoR, Hamsaz, etc.) coordinate all calls to these operations because of this potential conflict. Clearly, they are too conservative since only concurrent calls to `placeBid` and `closeAuction` that modify the *same* auction are conflicting, yet no calls to `placeBid` and `closeAuction` are allowed to run concurrently.

2.4 The ECRO Approach

Based on the observation that many conflicts are due to a "bad" ordering of *concurrent* operations, ECROs aim to solve those conflicts by reordering the operations, rather than coordinating them. We briefly categorize conflicts that occur in RDTs and explain how ECROs cope with them.

A first category of conflicts arises when replicas execute non-commutative operations concurrently, which are then exchanged and applied in different orders at different replicas, yielding diverged states. To ensure state convergence, ECROs deterministically order concurrent non-commutative operations at all replicas. In contrast, existing approaches [Houshmand and Lesani 2019; Kaki et al. 2018; Li et al. 2012, 2018, 2020; Sivaramakrishnan et al. 2015] coordinate these operations unnecessarily.

A second category of conflicts arises when some operation leads to a state transition, which makes concurrent operations unavailable in the new state (e.g. `closeAuction` closes an auction and may render concurrent `placeBid` operations unavailable). ECROs solve those conflicts by safely reordering unavailable operations before transitioning to the new state (e.g. reorder `placeBid` before `closeAuction`), whereas existing approaches coordinate those operations.

A third category of conflicts involves numeric invariants. For example, a banking application may implement the account balance as a non-negative counter (aka a bounded counter). However, concurrent withdrawals may overdraw the account and reordering the withdrawals does not solve this problem. For such conflicts, ECROs coordinate the problematic operations, and so does related work.

A final category of conflicts are due to replicas executing mutually exclusive operations concurrently. For example, if usernames must be unique then the `registerUser(username)` operation must take a lock on `username` to avoid that someone else registers the same username concurrently. These conflicts break invariants and thus require coordination between the operations. Like most approaches, ECROs coordinate mutually exclusive operations. Some applications may however allow temporary invariant violations to avoid coordination, and instead, repair the invariant after the facts [Guerraoui et al. 2016].

3 DERIVING SAFE SERIALIZATIONS FROM DISTRIBUTED SPECIFICATIONS

We previously explained that ECRO replicas compute a serialization of the method calls that respects application invariants and guarantees strong convergence. Key to efficiently enabling this is a static analysis that answers four questions:

(1) Which sequential method calls commute?
(2) Which concurrent method calls commute?
(3) When are concurrent method calls safe/unsafe?
(4) If two concurrent method calls are unsafe, does a safe ordering of the calls exist? If yes, which order?

To answer these questions, we developed Ordana: a static analysis tool that implements three analyses on distributed specifications. First, a dependency analysis (based on Houshmand and Lesani [2019]) detects dependencies between sequential method calls[3]. Second, a commutativity analysis (based on Balegas et al. [2015]) detects commutativity of concurrent calls. The dependency analysis and commutativity analysis are combined to detect commutativity of sequential calls. Finally, a novel safety analysis detects conflicts and finds solutions by reordering calls locally.

Before detailing the analyses, we define the components of an ECRO's distributed specification. To this end, we use RUBiS (cf. Section 2.3) as running example.

3.1 The ECRO Distributed Specification

Every ECRO data type consists of two parts: the data type's implementation and a distributed specification. The implementation encapsulates replicated state (e.g. class fields) and exposes a number of methods, some of which mutate the replicated state[4]. The distributed specification describes four aspects of every call to a mutating method m:

Context $\texttt{ctx}(m(\overline{a}), \sigma)$ Predicate that is known to be true when the method is first applied (with arguments $\overline{a}$) on the state σ of the origin replica, but need not necessarily be true when applied at remote replicas. For example, in RUBiS, users can only close auctions that are open. Therefore, if `closeAuction(auction1)` is generated in state σ, than `auction1` was open in σ (cf. Appendix A.2 in [De Porre et al. 2021]). However, this does not require `auction1` to still be open when the call is applied at remote replicas, as this would forbid replicas from closing `auction1` concurrently.

[3]Two method calls are sequential if one happened before the other (i.e. one was observed and only then was the other executed) [Lamport 1978]. If neither call happened before the other, we say that the calls are concurrent.
[4]Our implementation uses immutable collections. Methods return a modified copy of the state that replaces the old state.

Precondition $\texttt{pre}(m(\overline{a}), \sigma)$ Predicate that checks if m can be called with arguments $\overline{a}$ on state σ. Unlike the context, the precondition must be true before applying the call at *any* replica.

Postcondition $\texttt{post}(m(\overline{a}), \sigma_i, \sigma_j)$ Predicate describing the effects of applying m with arguments $\overline{a}$ on state σ_i which results in state σ_j. The predicate is true iff σ_j contains the effects of applying $m(\overline{a})$ on σ_i. For the sake of clarity, we may denote this as a state transition $\sigma_i \xrightarrow{m(\overline{a})} \sigma_j$.

Invariant $\texttt{inv}(m(\overline{a}), \sigma_i, \sigma_j)$ Predicate describing the behavior that is expected from (concurrent) calls to method m on state σ_i. The predicate is true iff the result state σ_j respects the invariants that are expected from applying m with arguments $\overline{a}$ on state σ_i. For example, we may want bids on an auction to get through even if the auction is closed concurrently. This can be expressed as follows:

$$\texttt{inv}(bid(user, auction, amount), \sigma_i, \sigma_j) = \begin{cases} \text{true}, & \text{if user's bid} \in \sigma_j \\ \text{false}, & \text{otherwise} \end{cases}$$

In the remainder of this section we explain how these four components are used by the various analyses to answer the aforementioned questions.

3.2 Dependency Analysis

This section details how to detect dependencies between method calls. We borrow the notion of dependency from Houshmand and Lesani [2019] and tailor it to meet ECROs' needs. Recall that users invoke methods on a replica and that method calls are propagated to all replicas. Method calls are allowed to execute at a replica only if their precondition holds.

Definition 3.1 (Correct call). A call c is correct in a given state σ iff its precondition holds in that state. Formally, $\forall c, \sigma \,.\, \texttt{correct}(c, \sigma) \iff \texttt{pre}(c, \sigma)$.

Definition 3.2 (Enabled call). A call c is enabled by a given state σ iff the context of c holds in σ and c is correct in σ. Formally, $\forall c, \sigma \,.\, \texttt{enabled}(c, \sigma) \iff \texttt{ctx}(c, \sigma) \wedge \texttt{correct}(c, \sigma)$. Two calls c_1 and c_2 are enabled by a state σ iff both are enabled by σ: $\forall c_1, c_2, \sigma \,.\, \texttt{enabled}(c_1, c_2, \sigma) \iff \texttt{enabled}(c_1, \sigma) \wedge \texttt{enabled}(c_2, \sigma)$.

Sequential calls may exhibit dependencies. Intuitively, a call c_2 depends on a call c_1 that happened before it if c_2 cannot execute before c_1.

Definition 3.3 (Independent and dependent calls). Let c_1 be a call that is enabled by state σ_0, σ_1 the state that results from executing c_1 on σ_0, and c_2 a call that is enabled by σ_1. We say that c_2 is independent of c_1 iff c_2 is correct in σ_0. Otherwise, c_2 depends on c_1, written as $dep(c_2, c_1)$. Formally, $\forall c_1, c_2 \,.\, \texttt{dep}(c_2, c_1) \iff \exists \sigma_0, \sigma_1 \,.\, \texttt{enabled}(c_1, \sigma_0) \wedge \texttt{post}(c_1, \sigma_0, \sigma_1) \wedge \texttt{enabled}(c_2, \sigma_1) \wedge \neg\texttt{correct}(c_2, \sigma_0)$.

Ordana's dependency analysis detects potential dependencies between pairs of methods and determines under which conditions these dependencies occur. Let $\langle m_1, m_2 \rangle$ be the method pair we want to analyze. To determine whether m_2 could depend on m_1, the analysis checks the satisfiability of the following formula using an SMT solver:

$$\exists \overline{a}_1, \overline{a}_2 \,.\, c_1 = m_1(\overline{a}_1) \wedge c_2 = m_2(\overline{a}_2) \wedge dep(c_2, c_1)$$

If the formula is unsatisfiable, this constitutes a proof that no call to m_2 exists that is dependent on a call to m_1. If it is satisfiable, a counter example exists in which a call to m_2 depends on a call to m_1. Ordana then restarts the analysis with equality relations between the methods' arguments to determine the root cause of this dependency. Although our approach cannot unravel

the cause of all dependencies, it works well in practice since dependencies often occur due to calls referring to an argument introduced by a previous call. For example, `bid(auction2,20)` depends on `open(auction1)` only when `auction2 = auction1`.

The dependency analysis returns a function $\mathsf{dep} :: \mathsf{C} \times \mathsf{C} \rightarrow \mathbb{B}$ that takes two calls and returns true if the first call depends on the second, false otherwise.

3.3 Concurrent Commutativity Analysis

Many RDTs leverage commutativity to ensure state convergence under concurrent method calls, without coordination. This led researchers to design static analyses capable of detecting methods that commute when executed concurrently, based on some specification [Balegas et al. 2015; Dimitrov et al. 2014; Gotsman et al. 2016; Kulkarni et al. 2011; Li et al. 2012].

However, commutativity is a property of method calls, not of concurrency. ECROs leverage commutativity for both concurrent and sequential calls. In this section, we focus on commutativity of concurrent calls, Section 3.4 elaborates on commutativity of sequential calls.

Definition 3.4 (Concurrent commutativity). Let σ_0 be some initial state, and c_1 and c_2 two concurrent method calls enabled by σ_0. We say that c_1 and c_2 *concurrent commute*, written as $c_1 \overset{\text{c}}{\rightleftharpoons} c_2$, iff applying the calls in either order leads to equivalent states. Formally, $\forall c_1, c_2 \,.\, c_1 \overset{\text{c}}{\rightleftharpoons} c_2 \iff \forall \sigma_0 \,.\, \mathsf{enabled}(c_1, c_2, \sigma_0) \land \forall \sigma_1, \sigma_2, \sigma_{12}, \sigma_{21} \,.\, \mathsf{post}(c_1, \sigma_0, \sigma_1) \land \mathsf{post}(c_2, \sigma_0, \sigma_2) \land \mathsf{post}(c_2, \sigma_1, \sigma_{12}) \land \mathsf{post}(c_1, \sigma_2, \sigma_{21}) \implies \mathsf{correct}(c_2, \sigma_1) \land \mathsf{correct}(c_1, \sigma_2) \land \sigma_{12} \equiv \sigma_{21}$.

To detect non-commutative method calls, Ordana analyzes all method pairs. For every method pair $\langle m_1, m_2 \rangle$, the analysis checks whether two concurrent calls to these methods exist that do not commute. To this end, it checks the satisfiability of the following formula using an SMT solver:

$$
\begin{gathered}
\exists \overline{a}_1, \overline{a}_2 \,.\, c_1 = m_1(\,\overline{a}_1\,) \land c_2 = m_2(\,\overline{a}_2\,) \land \\
\exists \sigma_0 \,.\, \mathsf{enabled}(c_1, c_2, \sigma_0) \land \\
\exists \sigma_1 \,.\, \mathsf{post}(c_1, \sigma_0, \sigma_1) \land \exists \sigma_2 \,.\, \mathsf{post}(c_2, \sigma_0, \sigma_2) \land \\
\exists \sigma_{12} \,.\, \mathsf{post}(c_2, \sigma_1, \sigma_{12}) \land \exists \sigma_{21} \,.\, \mathsf{post}(c_1, \sigma_2, \sigma_{21}) \land \\
\neg(\mathsf{correct}(c_2, \sigma_1) \land \mathsf{correct}(c_1, \sigma_2) \land \sigma_{12} \equiv \sigma_{21})
\end{gathered}
$$

If the above formula is unsatisfiable, this constitutes a proof that any two concurrent calls to m_1 and m_2, that are enabled by some initial state σ_0, are correct and commute. On the other hand, if it is satisfiable, concurrent calls to m_1 and m_2 exist that are not correct (i.e. cannot be applied one after the other) or do not commute. Ordana then restarts the analysis with equality relations between the calls' arguments to determine when this occurs. For example, concurrent `bid(auction1,10)` and `close(auction2)` calls always commute except when `auction1 = auction2`. The output of the analysis is a function $\mathsf{commutative} :: \mathsf{C} \times \mathsf{C} \rightarrow \mathbb{B}$ that takes two calls and returns true if the calls concurrent commute, false otherwise.

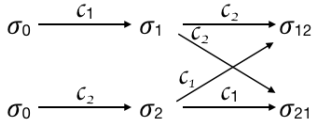


Fig. 3. State equivalence.

Note that we did not yet define state equivalence because it can be implemented in several ways. We may introduce a predicate that tests for state equivalence. However, this requires programmers to carefully define state equivalence and thus complicates the development of ECROs. Instead, Ordana derives state equivalence from the methods' postconditions. The states that result from applying the calls, in different orders, are equivalent iff they preserve the same effects: $\sigma_{12} \equiv \sigma_{21} \iff \mathsf{post}(c_2, \sigma_1, \sigma_{21}) \land \mathsf{post}(c_1, \sigma_2, \sigma_{12})$. This is

shown in Fig. 3, horizontal lines stand for sequential executions. The calls concurrent commute iff swapping their order leads to the same states (diagonal lines).

3.4 Deriving Sequential Commutativity

Sequential method calls differ from concurrent method calls because they can contain additional dependencies. If a call c_2 depends on a call c_1, then it follows that c_1 happened before c_2 (denoted $c_1 \prec c_2$). However, the inverse does not hold: causal relations do not necessarily imply dependencies (e.g. opening and closing different auctions). ECROs leverage this principle and allow sequential calls to be executed out of order if they commute and are independent; we say that these calls *sequentially commute*.

Definition 3.5 (Sequential commutativity). Let c_1 and c_2 be correct sequential calls such that $c_1 \prec c_2$. We say that c_2 sequentially commutes with c_1, written as $c_2 \overset{\text{S}}{\rightleftharpoons} c_1$, iff c_2 does not depend on c_1 and they concurrent commute, i.e. $\forall c_1, c_2 \,.\, c_2 \overset{\text{S}}{\rightleftharpoons} c_1 \iff \neg\mathsf{dep}(c_2, c_1) \wedge c_2 \overset{\text{C}}{\rightleftharpoons} c_1$.

Ordana derives sequential commutativity based on the dependency analysis and the concurrent commutativity analysis presented in Sections 3.2 and 3.3. Since dependencies between calls are asymmetric, sequential commutativity is also an asymmetrical relation. The output of Ordana is a function $\texttt{seqCommutative} :: \mathsf{C} \times \mathsf{C} \to \mathbb{B}$ that takes two calls and returns true if the first call sequentially commutes with the second call, false otherwise.

3.5 Safety Analysis

ECROs guarantee that no method call leaves the replicas in a conflicting state, i.e. a state that violates the invariants defined by the data type's distributed specification. To this end, Ordana implements a safety analysis that detects pairs of concurrent methods that could infringe invariants (similar to Balegas et al. [2015]; Gotsman et al. [2016]; Houshmand and Lesani [2019]) and introduces a novel technique to find solutions to these conflicts without imposing coordination. Before delving into the analysis, we define safety.

Definition 3.6 (Safe calls). Two concurrent method calls are safe iff applying them in *any* order preserves the methods' preconditions and invariants, otherwise, they are unsafe.

Definition 3.7 (Safe methods). Two methods are safe iff all pairs of correct concurrent calls to those methods are safe.

Definition 3.8 (Safe serialization). A serialization of correct calls is safe iff all pairs of concurrent calls are safe or the ordering of concurrent calls preserves their invariants.

To identify unsafe methods, Ordana analyzes the invariants of all method pairs, and checks if some serialization of concurrent calls to these methods could violate the invariants. Given a method pair $\langle m_1, m_2 \rangle$, the safety analysis checks the satisfiability of the following formula:

$$\begin{gathered}
\exists \overline{a}_1, \overline{a}_2 \,.\, c_1 = m_1(\,\overline{a}_1\,) \wedge c_2 = m_2(\,\overline{a}_2\,) \wedge \\
\exists \sigma_0 \,.\, \mathsf{enabled}(c_1, c_2, \sigma_0) \wedge \\
\exists \sigma_1 \,.\, \mathsf{post}(c_1, \sigma_0, \sigma_1) \;\wedge \exists \sigma_{res} \,.\, \mathsf{post}(c_2, \sigma_1, \sigma_{res}) \wedge \\
\neg(\,\mathsf{pre}(c_2, \sigma_1) \wedge \mathsf{inv}(c_1, \sigma_0, \sigma_{res}) \wedge \mathsf{inv}(c_2, \sigma_1, \sigma_{res})\,)
\end{gathered}$$

If the above formula is unsatisfiable, any two calls, c_1 to method m_1 and c_2 to method m_2, that are enabled by some initial state σ_0 preserve the calls' preconditions and invariants when applied one after the other ($c_1 \prec c_2$) on σ_0. This constitutes a proof that c_1 followed by c_2 is a safe serialization. On the other hand, if the formula is satisfiable, applying c_1 and c_2 in order on σ_0 violates a precondition

or an invariant. Ordana then restarts the analysis with equality relations between the arguments to determine the cause of the conflict.

The output of the described safety analysis are two functions: `restrictions` :: `C` $\rightarrow R$ and `resolution` :: `C`×`C` $\rightarrow \{<, >, \top, \bot\}$. The former, `restrictions`, takes a call c and returns a set R of restrictions. These are all the methods that require coordination because they may violate invariants when executed concurrently with c and no safe serialization exists. Restrictions correspond to the locks that the ECRO algorithm takes before executing call c and include the argument of c that causes the conflict (if detected). Consider again the RUBiS application, concurrent calls to `registerUser` may violate the invariant that usernames must be unique. Therefore, the analysis places a restriction on `registerUser` that locks the username passed to the call.

The latter function, `resolution`, takes two concurrent calls and returns $\top$ if the calls are safe. If the calls (say c_1 and c_2) are unsafe but a safe serialization exists it will return an ordering of the calls ($c_1 < c_2$ or $c_1 > c_2$) that is safe. Otherwise, it returns $\bot$ since the calls require coordination, i.e. $\texttt{restrictions}(c_1) \neq \emptyset \vee \texttt{restrictions}(c_2) \neq \emptyset$.

4 EXPLICITLY CONSISTENT REPLICATED OBJECTS

We now formally define ECROs and explain how their replication algorithm uses the information inferred by Ordana to serialize method calls safely while minimizing coordination.

We represent an ECRO as a tuple $\langle \Sigma, \sigma_0, \texttt{M}, \texttt{G}, \texttt{t}, \texttt{F} \rangle$, where Σ is the set of possible states, σ_0 is the initial state, `M` is the set of methods, `G` is the object's execution graph, `t` is the current topological order of graph `G`, and `F` is the set of functions produced by Ordana (cf. Section 3). The execution graph $\texttt{G} = \langle \texttt{C}, \texttt{E} \rangle$ is a labeled directed acyclic graph (DAG) where vertices (`C`) are method calls, and edges (`E`) express relations between calls. The algorithm operates on this graph. In a nutshell, it considers three types of edges:

happened-before edges (hb-edges) enforce causality. For every pair of causally related calls a corresponding hb-edge is added to the graph if they do not commute. Causal relations between sequentially commutative calls are ignored since their order does not affect the outcome.

conflict-order edges (co-edges) enforce the invariants defined in the distributed specification (i.e. safety). For every two unsafe concurrent calls, the algorithm checks if a safe serialization of the calls exists. If that is the case, the corresponding co-edge is added between the calls. Consider again the Add-Wins Set from Section 2.2. The algorithm adds a co-edge from `remove(x)` to concurrent `add(x)` calls since Ordana proves that applying `remove(x)` before `add(x)` guarantees add-wins semantics.

arbitration order edges (ao-edges) enforce state convergence. In some cases, concurrent calls are safe but do not commute. All replicas must execute those calls in the same order to guarantee strong convergence. Replicas order these calls deterministically by adding an ao-edge between them, whose direction is based on the globally unique identifiers of the calls.

By combining these three types of edges, any topological ordering of graph `G` is a safe serialization that preserves dependencies and guarantees strong convergence. Several topological orders may exist because the algorithm does not add edges between safe calls that commute. As we prove in Section 4.2, different topological orderings only interchange commutative calls and thus lead to equivalent states.

4.1 Replication Algorithm

Algorithm 1 presents an overview of the replication algorithm that is run by each ECRO replica. When the user invokes a method on a replica, it is handled by the replica's `execute_local` function, and later integrated at remote replicas using the `execute_remote` function.

Algorithm 1 ECRO replication algorithm main functions

```
1:  ⟨Σ, σ0, M, G, t, F⟩, with G = ⟨C, E⟩                          ▷ ECRO's internal state
2:  σ: Σ                                                          ▷ object current state σ
3:  function EXECUTE_LOCAL(m(ā))          ▷ execution of method m with parameters ā, at origin replica
4:      c ← ⟨ m(ā), uniqueId(), timestamp() ⟩                     ▷ tag method call with unique id and logical timestamp
5:      if restrictions(c) ≠ ∅ then                               ▷ call c may be unsafe
6:          acquire_locks(restrictions(c))
7:      C ← C ∪ { c }                                             ▷ add call c to the graph vertices
8:      for v ∈ C ∧ v ≠ c do                                      ▷ determine relevant hb-edges for call c
9:          if not seqCommutative(c, v) then                      ▷ call c is sequential non-commutative with call v
10:             E ← E ∪ {⟨v, hb, c⟩}                              ▷ add hb-edge between call v and call c
11:     t ← t + c                                                 ▷ local call c has no impact on topological order
12:     σ ← apply(σ, c)                                           ▷ execute call c on current state σ
13:     commitStableCalls()                                       ▷ commits previous calls if there is a single replica
14:     propagate(c)          ▷ propagation of call c to remote replicas (at-least-once causal delivery)
15:     if hasLocks() then
16:         wait_ack()                                            ▷ if needed, wait for ack
17:         release_locks(restrictions(c))
18: function EXECUTE_REMOTE(c)                                    ▷ execution of call c at remote replica
19:     C ← C ∪ { c }                                             ▷ add call c to the graph vertices
20:     for v ∈ C ∧ v ≠ c do                                      ▷ determine relevant edges (relations) for call c
21:         if v ≺ c ∧ not seqCommutative(c, v) then              ▷ call c is sequential non-commutative with call v
22:             E ← E ∪ { ⟨v, hb, c⟩}                             ▷ add hb-edge between call v and call c
23:         else if v ∥ c then                                    ▷ call v is concurrent with call c
24:             if resolution(c, v) = < then                      ▷ conflict solved by ordering c before v
25:                 E ← E ∪ { ⟨c, co, v⟩}                         ▷ add co-edge between call c and call v
26:             else if resolution(c, v) = > then                 ▷ conflict solved by ordering v before c
27:                 E ← E ∪ { ⟨v, co, c⟩}                         ▷ add co-edge between call v and call c
28:             else if resolution(c, v) = ⊤ ∧   ▷ calls c and v are non-conflicting and non-commutative
29:                     not commutative(c, v) then
30:                 if Id(c) < Id(v) then                         ▷ arbitrate a deterministic order based on ids
31:                     E ← E ∪ { ⟨v, ao, c⟩}                     ▷ add ao-edge between call c and call v
32:                 else E ← E ∪ { ⟨v, ao, c⟩}                    ▷ add ao-edge between call v and call c
33:     t ← dynamicTopologicalSort(G)          ▷ apply algorithm to subgraph of concurrent calls to c
34:     σ ← apply(σ0, t)                                          ▷ execute calls on initial state σ0
35:     commitStableCalls()                                       ▷ commit prefix of causally stable calls
```

Local Method Calls. Upon receiving a local request to execute method m with arguments $\overline{a}$, the `execute_local` function creates a new call c containing the method and its arguments $m(\overline{a})$, a globally unique identifier[5], and a logical timestamp[6]. If the call is unsafe, it is coordinated by acquiring the necessary locks (line 6). The `restrictions` function (returned by the safety analysis) leverages the call's arguments to ensure the right lock granularity. Since c is a new local request, all calls already contained by the replica's execution graph happened before c. Thus, call c is added to the graph and an hb-edge is added between c and every call that does not sequentially commute

[5]We use Lamport clocks [Lamport 1978] to generate globally unique identifiers as they define a total order of calls.
[6]Any logical timestamp that can track causality can be used.

Algorithm 2 Committing causally stable calls

```
1:  ⟨Σ, σ0, M, G, t, F⟩, with G = ⟨C, E⟩                ▷ ECRO's internal state
2:  function commitStableCalls
3:      stablePrefix ← true
4:      i ← 0                                            ▷ number of causally stable calls
5:      while i < |t| ∧ stablePrefix do                  ▷ iterate over a prefix of stable calls
6:          call ← t[i]
7:          stablePrefix ← isStable(call)                ▷ determine if the call is stable based on its vector clock
8:          if stablePrefix then
9:              σ0 ← apply(σ0, call)                     ▷ update initial state
10:             C ← C \ { call }
11:             E ← (E \ in(call)) \ out(call)           ▷ remove incoming and outgoing edges
12:             i ← i + 1
13:     t ← drop(i, t)                                   ▷ remove the prefix of stable calls from the topological order
```

with c (line 10); these edges do not affect the topological ordering. Next, local call c is appended to the end of the current topological order (line 11), c is applied on the current state σ (line 12), and causally stable calls are committed to keep the graph small (line 13) as will be explained later. Lastly, the call is propagated to the other replicas and acquired locks are released after receiving confirmation from remote replicas that the call has been applied (line 17).

Integrating Remote Method Calls. Upon receiving a (safe or unsafe) remote call c, the `execute_remote` function adds c to the vertices (line 19) and adds the necessary edges to the graph (lines 20-32). For calls that happened before c the approach is the same as the one described before for local calls. For concurrent calls (line 23) we distinguish two cases. In the first case, calls c and v are unsafe, but a safe serialization exists. The algorithm then uses the resolution function returned by the safety analysis to determine the direction of the `co`-edge (i.e. how to order calls, lines 24-27). In the second case, calls c and v are safe but do not commute (line 28). To ensure convergence, the function uses the calls' identifiers to deterministically add an `ao`-edge between c and v. A dynamic topological sort is performed on the execution graph to recompute only the subgraph that changed (line 33); these changes are limited to calls that are concurrent to c. Line 34 updates the state by applying, in order, the calls from the topological order on the initial state. Finally, on line 35, causally stable calls are committed to keep the execution graph small. Although not shown in the algorithm, if c is an unsafe call an acknowledgment is sent to the origin replica.

Optimizations. In order to keep the algorithm efficient and avoid replying the entire operation history for every incoming call, two optimizations were applied to the ECRO algorithm. First, causally stable calls are committed to keep the graph small, as shown in Algorithm 2. Second, snapshots of intermediate states are kept to minimize the calls that have to be recomputed. We now briefly explain how these optimizations work.

A call c is *causally stable* [Baquero et al. 2017] at a replica r if r knows that all other replicas also observed c. Causal stability can be derived from the vector clocks carried by calls when they are propagated to replicas; a call c with vector clock vc is causally stable at replica r, if vc happened before or is equal to the latest clock received from every other replica[7]. Based on this observation, r knows that no more calls that are concurrent to c can arrive. Replicas can thus - locally and without requiring coordination - commit a prefix of causally stable calls as their positions are

[7]Replicas can periodically send a no-op to ensure that calls stabilize even if some replicas do not generate calls.

known to be fixed within the serialization. This is described by the `commitStableCalls` function in Algorithm 2. The *initial* state σ_0 is updated by applying the longest prefix of stable calls (line 9), whereafter, those calls and their incoming and outgoing edges can safely be removed from the graph (line 10 and 11). However, even if replicas commit causally stable calls, they need to replay the entire serialization of calls to compute the current state (line 34 in Algorithm 1). To address this issue, replicas take snapshots of intermediate states which enables efficient rollbacks to prior states. For example, if the topological sort (line 33) results in $t = t_1.t_2$ where t_1 is unchanged and t_2 is the part of the serialization that changed, then the replicas can roll back to the snapshot of the state after t_1 such that only the calls in t_2 need to be replayed[8]. Note that if there is no snapshot available that corresponds to the state after t_1, the algorithm needs to roll back to an older snapshot (ideally the one that corresponds to the longest prefix of t_1). In ECROs, programmers can configure the "snapshot interval", i.e. after how many calls a snapshot must be taken. This interval is a trade-off between latency and memory. The more snapshots replicas take, the less calls they need to replay but the more memory they use. The less snapshots replicas take, the less memory they consume but the more calls need to be replayed. We argue that the snapshot interval should be smaller (i.e. take more snapshots) if operations are costly and bigger if operations are fast.

Cycles. To keep the graph acyclic, we implement an efficient and deterministic approach that detects and solves cycles. We briefly outline this approach and refer to Appendix B in [De Porre et al. 2021] for the complete specification. If a newly added edge $c_1 \longrightarrow c_2$ causes a cycle, at least one path from c_2 to c_1 exists. The algorithm computes all paths from c_2 to c_1 and breaks them by removing one ao-edge on each path. These edges can be removed without putting convergence at risk as they impose an artificial ordering between non-commutative calls. Hence, we solved the cycle while keeping all non-commutative calls ordered. Occasionally, the cycle is caused by a combination of hb-edges and co-edges. These cannot be removed without violating convergence and safety. Instead, the algorithm deterministically discards a call that breaks the cycle. Information about discarded ao-edges and calls is propagated between replicas to ensure that all replicas eliminate the same ao-edges and/or calls and thus still converge. Note that discarding the operation that causes a cycle may cause an anomaly observed by a client. Future work could explore alternative ways to discard operations.

Consistency Guarantees. ECROs guarantee Explicit Consistency [Balegas et al. 2015] which strengthens eventual consistency with application-level invariants. To this end, Algorithm 1 ensures that every replica applies a serialization of the calls that respects dependencies between calls, totally orders non-commutative calls, and upholds application-level invariants.

4.2 Algorithm Correctness

We now prove that ECROs guarantee convergence and safety with respect to the distributed specification of the data type. Because the replication algorithm does not order pairs of calls that commute and are safe, several topological orders of the graph may exist. We show that all topological orderings are safe (Theorem 4.4) and that convergence is guaranteed for replicas that received the same method calls (Theorem 4.5).

Lemma 4.1. *Two replicas of an ECRO that observed the same calls have the same execution graph:*

$$\forall r_1 = \langle \Sigma, \sigma_0, \mathsf{M}, \mathsf{G}_1, \mathsf{t}_1, \mathsf{F} \rangle,\ r_2 = \langle \Sigma, \sigma_0, \mathsf{M}, \mathsf{G}_2, \mathsf{t}_2, \mathsf{F} \rangle.$$
$$\mathsf{G}_1 = \langle \mathsf{C}_1, \mathsf{E}_1 \rangle \land \mathsf{G}_2 = \langle \mathsf{C}_2, \mathsf{E}_2 \rangle \land \mathsf{C}_1 = \mathsf{C}_2 \implies \mathsf{E}_1 = \mathsf{E}_2 \implies \mathsf{G}_1 = \mathsf{G}_2$$

[8]Note that dynamic topological sorting algorithms can return the index of the first change in the topological order such that we do not have to compute it manually based on the old ordering, which would be costly.

Proof. By case analysis on the edges added to the graphs for each call. The complete proof is in Appendix B.2 in [De Porre et al. 2021]. □

Definition 4.2 (Equivalent serializations). Two serializations t_1 and t_2 of a set of method calls C are equivalent iff every pair of non-commutative calls appears in the same order in both t_1 and t_2:

$$\forall t_1, t_2 \,.\, t_1 \equiv t_2 \iff \forall c_1, c_2 \in \mathtt{C} \,.\, \neg \mathtt{commutative}(c_1, c_2) \implies$$
$$(t_1[c_1] < t_1[c_2] \iff t_2[c_1] < t_2[c_2]) \land (t_1[c_1] > t_1[c_2] \iff t_2[c_1] > t_2[c_2])$$

where $t[c]$ returns the position of call c in serialization t.

Definition 4.3 (Equivalent replicas). Two replicas r_1 and r_2 of an ECRO are equivalent iff they have the same execution graph G (i.e. observed the same calls) and their topological orderings of G are equivalent: $\forall r_1 = \langle \Sigma, \sigma_0, \mathtt{M}, \mathtt{G}_1, \mathtt{t}_1, \mathtt{F} \rangle, r_2 = \langle \Sigma, \sigma_0, \mathtt{M}, \mathtt{G}_2, \mathtt{t}_2, \mathtt{F} \rangle \,.\, r_1 \equiv r_2 \iff \mathtt{G}_1 = \mathtt{G}_2 \land t_1 \equiv t_2$.

Theorem 4.4. *All topological orderings of an execution graph* G *of an ECRO replica are safe serializations.*

Proof. By induction on the length of the topological ordering.
Base case. In the base case no calls occurred, so the topological ordering is empty and trivially safe.
Induction step. Assume replica r_1 has a topological order of dimension n that is safe. If a new call c is executed at replica r_1 two cases are possible: either c is a local or a remote call.

Case 1. If c is a local method call we distinguish two new cases. In the first case, the call is unsafe and the analysis tells us which method calls are conflicting. Algorithm 1 uses this information to coordinate the call such that no unsafe call can execute concurrently, thereby guaranteeing safety. In the second case, we know from the analysis that c is safe with respect to all other possible calls. Hence, we can execute c and the resulting topological order(s) are safe serializations.

Case 2. If call c was propagated by replica r_2, then its execution was locally safe at replica r_2. We distinguish three cases. In the first, the call is unsafe and the analysis did not find a solution. The originating replica r_2 then coordinated the call such that no conflicting call can execute concurrently, thereby, guaranteeing safety. In the second case, the call is unsafe but the analysis found a solution. If a conflicting call occurs concurrently to c, all replicas add the same co-edge between those calls. This co-edge guarantees safety since the analysis proved that this ordering preserves the invariants. In the third case, the call is safe with regard to all possible correct concurrent calls and thus cannot violate safety.

We showed that starting from an execution graph with a safe topological order of dimension n and a new call c, Algorithm 1 builds an expanded graph whose topological order of dimension $n + 1$ is also a safe serialization. □

Theorem 4.5. *Two ECRO replicas that observed the same calls* C *converge to equivalent states. Formally:* $\forall r_1 = \langle \Sigma, \sigma_0, \mathtt{M}, \langle \mathtt{C}_1, \mathtt{E}_1 \rangle, \mathtt{t}_1, \mathtt{F} \rangle, r_2 = \langle \Sigma, \sigma_0, \mathtt{M}, \langle \mathtt{C}_2, \mathtt{E}_2 \rangle, \mathtt{t}_2, \mathtt{F} \rangle \,.\, \mathtt{C}_1 = \mathtt{C}_2 \implies r_1 \equiv r_2$

Proof. Since both replicas observed the same calls, we know from Lemma 4.1 that they have the same execution graph, $G_1 = G_2$. This graph is constructed by successive applications of Algorithm 1, hence, sequential non-commutative calls are ordered by hb-edges and concurrent non-commutative calls are ordered by co-edges and ao-edges. Any topological ordering of the graph thus maintains the relative order of non-commutative calls. From Definitions 4.2 and 4.3 it follows that the replicas converge to equivalent states. The complete proof is in Appendix B.2 in [De Porre et al. 2021]. □

5 IMPLEMENTATION

We now describe our prototype implementation of the ECRO approach in Scala. Ordana, our analysis tool, consists of two parts, a parser and an analyzer. The parser is based on Indigo [Balegas

Table 1. Portfolio of ECRO data types and their description.

Data Type	Description and distributed semantics
Counter	Supports increments and decrements.
EW-Flag	Flag that can be enabled and disabled. Guarantees enable-wins semantics in case the flag is enabled and disabled concurrently.
DW-Flag	Similar to EW-Flag but guarantees disable-wins semantics.
AW-Set	Wrapper around Scala's built-in immutable set. Provides add-wins semantics similar to the OR-Set CRDT [Shapiro et al. 2011a].
RW-Set	Similar to AW-Set but provides remove-wins semantics.
AW-Map	Wrapper around Scala's built-in immutable map. Values can be complex objects and are updated by overriding the key with the new value. Provides add-wins semantics when the same key is added and removed concurrently, and last-writer-wins semantics for concurrent adds of the same key.
RW-Map	Similar to AW-Map but provides remove-wins semantics when a key is added and removed concurrently.
Stack	Stack allowing push, pop, and top operations. Push operations execute optimistically and are totally ordered. Pop operations are coordinated in order not to pop more elements than there are on the stack.
Queue	Wrapper around Scala's built-in immutable queue. Enqueue operations run optimistically and are totally ordered. Dequeue operations are coordinated to avoid dequeueing more elements than there are in the queue.
List	Provides operations to prepend, insert, and delete elements, and to map a function over the list.
RUBiS	eBay-like auction system similar to the RUBiS benchmark [Cecchet and Marguerite 2009].

et al. 2015] and translates the first-order logic formulas from an ECRO's distributed specification to Z3 formulas. The analyzer implements the analyses presented in Section 3, and executes them using a Java binding for Z3. The results of Ordana are then passed to the ECRO replication algorithm.

The implementation of the replication algorithm uses a dynamic topological sort algorithm [Pearce and Kelly 2006] provided by the JGraphT library[9] and takes snapshots of intermediate states to efficiently roll back concurrent calls when they are reordered. Snapshots are garbage collected once their state is stable. We also remove calls from the replica's execution graph once they are *causally stable* [Baquero et al. 2017]. This is safe since no more concurrent calls can arrive, i.e. the order of the call in the serialization is stable across all replicas. We integrated our ECRO implementation in Squirrel [De Porre and Gonzalez Boix 2019], a distributed in-memory key-value (DKV) store for Scala. Programmers can implement custom ECROs and store them in Squirrel which automatically replicates them across all copies of the store and propagates method calls.

5.1 Portfolio of ECRO Data Types

We now present a portfolio of RDTs that we implemented with ECROs and integrated in Squirrel[10]. Our portfolio covers existing RDTs (counters, flags, sets, maps, lists), new RDTs for which no prior (C)RDT design exists as they require coordination (stacks and queues), and a geo-distributed RUBiS

[9] https://jgrapht.org/

[10] The complete portfolio of ECROs is included in our software artifact at https://doi.org/10.5281/zenodo.5410793.

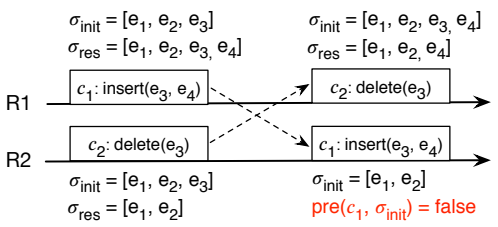
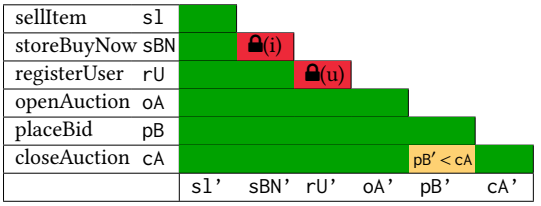


Fig. 4. Conflict that requires R2 to reorder the calls.

Table 2. Outcome of Ordana's safety analysis for RUBiS. 🔒(i) = lock(item), 🔒(u) = lock(user), and pB' < cA when auction = auction'.



application that is built from sequential data types (i.e. without having to devise ad-hoc RDTs). Table 1 provides an overview of all the data types included in the portfolio, accompanied by a brief description. Due to space constraints we only elaborate on the list and RUBiS data types; sets have already been discussed in Section 2.2.

List. The list data type provides methods to prepend elements to the list, insert elements after other elements in the list, delete elements from the list, and map functions over the list. We added a precondition to `insert` to ensure that the element after which to insert (called the reference element) exists. We also added an invariant to `insert` to ensure that the inserted element occurs in the resulting list and is not overwritten by a concurrent `map`. All methods are allowed to run optimistically, i.e. they do not require coordination. If two elements are inserted at the same position concurrently, the algorithm totally orders them across all replicas as those calls do not commute.

Concurrent insertions and deletions may lead to conflicts. Figure 4 shows the case where replica R1 inserts a new element e_4 behind e_3 while concurrently replica R2 deletes e_3. After exchanging the calls, R2 cannot insert e_4 because e_3 is no longer present in the list, thereby violating insert's precondition. R2 can solve this conflict by reordering the calls such that e_4 is inserted before deleting e_3. As detected by Ordana, this reordering is safe and only needed when deleting the reference element (e_3 in the previous example) of one or more concurrent insertions. Similarly, map operations are reordered before concurrent insertions in order not to modify newly inserted elements (as this would violate the invariant of `insert`).

RUBiS. Our RUBiS data type is the eBay-like auction system introduced in Section 2.3. The RUBiS ECRO provides methods for registering users, selling items, buying items, opening auctions, bidding on auctions, and closing auctions. Users must be unique, the stock of an item may not go below zero, and bids can only be placed on open auctions. Table 2 shows the result of the safety analysis. The upper triangle of the matrix is omitted as the relations are symmetrical. Most method pairs are safe (colored green). `placeBid` and `closeAuction`, however, do not commute and may lead to conflicts when a bid is placed on an auction that is closed concurrently (colored orange). Ordana found a solution to this conflict by ordering `placeBid` calls before concurrent `closeAuction` calls, and can uphold the invariants without coordination. Finally, `storeBuyNow` and `registerUser` are unsafe (colored red) because concurrent calls may lead to a negative stock or duplicate users. These conflicts cannot be avoided by reordering the calls, hence, ECROs coordinate them. To buy an item or register a user, the replica must first acquire a lock on the given item or user. This lock *only* restricts buying/registering the same item/user concurrently.

6 EVALUATION

We now evaluate our work to check whether the ECRO approach is suitable for building geo-distributed applications. Throughout this evaluation we assess the practicality and performance of our approach. We conduct several experiments to answer the following research questions:

RQ1. Do ECROs aid the development of RDTs compared to state-of-the-art approaches?
RQ2. Can the static analyses, described in Section 3 and included in Ordana, be used in practice?
RQ3. Does the ECRO algorithm scale?
RQ4. How do ECRO-enabled geo-distributed applications perform compared to other approaches?

6.1 Methodology

Performance experiments reported in this section were conducted on Amazon EC2 `m5.xlarge` virtual machine instances. Each VM has 4 virtual CPUs and 16GiB of RAM. All benchmarks are implemented using JMH [OpenJDK [n. d.]], a benchmarking library for the JVM that helps avoiding common pitfalls, such as loop optimizations and dead code elimination [Ponge 2014]. Each benchmark starts with a warmup phase, followed by the actual measurement phase consisting of 20 iterations. To avoid run-to-run variance we use the default setting of 5 JVM forks, which repeats the benchmark 5 times in fresh JVMs. This yields a total of 100 samples per benchmark.

6.2 RQ1: Evaluating the Design of RDTs using ECROs

We now evaluate the impact of ECROs on the design and implementation of applications involving RDTs. We first compare the implementation of replicated sets with ECROs against well-known CRDT implementations [Shapiro et al. 2011a]. Then, we compare the RUBiS system (cf. Section 2.3) with ECROs against existing solutions such as PoR [Li et al. 2018] and RedBlue [Li et al. 2012].

6.2.1 Replicated Sets. We now compare the design and implementation of sets implemented with operation-based CRDTs and ECROs. Operation-based CRDTs split every operation in two phases: a first phase that prepares a message to be broadcast to every replica including itself (`prepare` method), and a downstream phase that applies such incoming messages (`downstream` method).

Listing 4 shows the implementation of an OR-Set CRDT in Scala, which ensures add-wins semantics by associating a globally unique tag to every element it adds (lines 5-8). When some replica removes an element, it tells all replicas to remove only the tags it observed for that element (line 9). An element is part of the set if its set of tags is non-empty (line 3). Since messages are delivered in causal order and replicas cannot remove the tags of elements that are added concurrently, the OR-Set guarantees add-wins semantics.

Providing remove-wins set semantics requires a completely different CRDT implementation. Listing 5 shows the implementation of a 2P-Set CRDT in Scala, which ensures remove-wins semantics by keeping two grow-only sets: `added` and `removed` (line 1). Elements are added by adding them to the `added` set and removed by adding them to the `removed` set (lines 4 and 6). An element is considered in the set if it is in the `added` set and not in the `removed` set (line 2). A consequence of this design is that a removed element can never be added again.

With ECROs, developers change the semantics of the set by modifying its specification instead of its implementation. This enables (1) RDT implementations to be reused, and (2) RDTs can exhibit different distributed semantics based on the application's needs, without rethinking the data type. As shown in Section 2.2, the add-wins and remove-wins set ECROs share the same sequential implementation (cf. Listing 1) and only differ in their specification (cf. Listing 2): the former associates an invariant to the `add` operation that guarantees add-wins semantics, while the latter associates an invariant to the `remove` operation that guarantees remove-wins semantics.

```
case class Tag[ID](replica: ID, ctr: Int)
case class ORSet[V, ID](myID: ID, counter: Int, elements: Map[V, Set[Tag[ID]]]) {
  def contains(e: V) = elements.getOrElse(e, Set.empty[Tag[ID]]).nonEmpty
  def prepareAdd(e: V): (V, Tag[ID]) = (e, Tag(myID, counter + 1))
  def addDownstream(tup: (V, Tag[ID])) = {
    val (e, tag) = tup; val tags = elements.getOrElse(e, Set())
    val newCounter = if (tag.replica == myID) tag.ctr else counter
    ORSet(myID, newCounter, elements + (e -> (tags + tag))) }
  def prepareRemove(e: V): (V, Set[Tag[ID]]) = (e, elements.getOrElse(e, Set()))
  def removeDownstream(tup: (V, Set[Tag[ID]])) = {
    val (e, tags) = tup; val observedTags = elements.getOrElse(e, Set())
    ORSet(myID, counter, elements + (e -> observedTags.diff(tags)))
  }
}
```

Listing 4. Implementation of an Observed-Removed Set (OR-Set) CRDT in Scala.

```
case class TwoPSet[V](added: Set[V] = Set(), removed: Set[V] = Set()) {
  def contains(element: V) = added.contains(element) && !removed.contains(element)
  def preAdd(element: V) = element
  def addDownstream(element: V) = TwoPSet(added + element, removed)
  def preRemove(element: V) = element
  def remove(element: V) = TwoPSet(added, removed + element)
}
```

Listing 5. Implementation of a Two-Phase Set (2P-Set) CRDT in Scala.

6.2.2 RUBiS. We now compare the implementation of RUBiS with ECROs (cf. Section 2.3) against its implementation with two state-of-the-art solutions: PoR and RedBlue consistency. We do not provide code snippets due to space constraints.

RedBlue and PoR require programmers to manually identify all potential conflicts in the application and determine restrictions that guarantee state convergence and invariant preservation. Table 4 shows that RedBlue requires 10 restrictions for the RUBiS system because it coordinates all unsafe shadow operations. On the other hand, PoR determines only 3 restrictions for RUBiS. ECROs, instead, automatically derive a *minimal* set of restrictions based on the results of the safety analysis shown in Table 2. This results in only two restrictions (also shown in Table 4) because Ordana finds a solution for the conflict between `placeBid` and `closeAuction` as mentioned in Section 5.1.

Sieve [Li et al. 2014] automatically derives the restrictions for RedBlue consistency, based on a first-order logic specification of the operations. In contrast, ECROs can derive fine-grained restrictions thanks to its novel safety analysis and *only* restricts buying/registering the same item/user concurrently. Moreover, specifications in Sieve (as well as in other automatic solutions [Balegas et al. 2015; Gotsman et al. 2016; Kaki et al. 2018; Sivaramakrishnan et al. 2015]) are plain strings containing first-order logic formulas, while ECROs provide an internal DSL for first-order logic in Scala. Our DSL simplifies the development of the specifications since (1) it provides support for common tasks (e.g. copying relations between states, expressing uniqueness constraints), (2) syntax errors and type errors are caught by the compiler, and (3) programmers can leverage Scala's existing abstraction and modularisation mechanisms (e.g. classes, traits, etc.).

Table 3. Average time for Ordana to analyze ECRO specifications on an Amazon EC2 `m5.xlarge` instance.

	Counter	EW-Flag	DW-Flag	AW-Set	RW-Set	AW-Map	RW-Map	Stack	Queue	List	RUBiS
Time (ms)	58	67	73	93	95	120	117	199	175	1732	4175

Table 4. Restrictions over the RUBiS operations enforced by RedBlue and PoR, taken from Li et al. [2018] and extended with ECROs.

RedBlue consistency	PoR consistency
r(registerUser, registerUser) r(storeBuyNow, storeBuyNow) r(placeBid, placeBid) r(closeAuction, closeAuction) r(placeBid, closeAuction) r(registerUser, storeBuyNow) r(registerUser, placeBid) r(registerUser, closeAuction) r(storeBuyNow, placeBid) r(storeBuyNow, closeAuction)	r(registerUser, registerUser) r(storeBuyNow, storeBuyNow) r(placeBid, closeAuction) **ECRO** r(registerUser, registerUser, <user>) r(storeBuyNow, storeBuyNow, <item>)

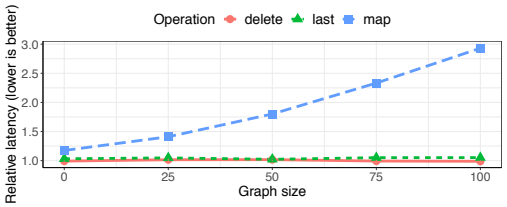


Fig. 5. Latency of operations on an ECRO list. We disabled the JIT compiler to better show the impact of the graph's size on the ECRO algorithm.

6.3 RQ2: Evaluating Ordana, Our Static Analysis Tool

We now measure the execution time of Ordana on the distributed specification of each data type in our portfolio of ECROs, presented in Section 5. The results, presented in Table 3, show that most data types are statically analyzed in less than 200 ms. This results from the fact that their specifications are rather simple and concise. The execution time for the list data type and RUBiS applications are considerably higher. For lists, this is due to the complexity of the specification as operations manipulate next pointers. For RUBiS, the higher execution time comes from the fact that it has a bigger interface and thus more operations to analyze.

Based on these results, we conclude that Ordana is suited to analyze the distributed specifications of ECROs, as even the RUBiS application is analyzed in less than 5 seconds. Note that the analyses run at compile time and *only* reanalyze the distributed specifications that changed, enabling their adoption within integrated development environments.

6.4 RQ3: Evaluating the Scalability of the ECRO Algorithm

The ECRO algorithm maintains a directed acyclic graph of tentative operations. We now evaluate the scalability of the algorithm with regard to the size of the graph (i.e. the number of causally unstable operations [Almeida et al. 2015]) for three types of operations: (1) side-effect free operations, (2) operations that are safe and commute, and (3) unsafe operations that do not commute.

6.4.1 Comparison to Sequential Data Types. We now measure the latency of three operations (`last`, `delete`, and `map`) on an ECRO list, which tipify the aforementioned types of operations. The benchmark runs on a single Amazon EC2 m5.xlarge instance and measures the latency of the three operations on an ECRO list containing 50K elements that is constructed by successive insertions: $insert(e_1, e_2); insert(e_2, e_3); \ldots; insert(e_{n-1}, e_n)$. We use Scala's built-in list as baseline and normalize the measurements. Figure 5 shows the relative latency of each operation.

Last. Returns the last element of the list. Since the operation has no side-effects, it executes immediately (no need to add it to the graph). As a result, we observe no significant performance difference compared to Scala's built-in list; the relative latency is approximately 1.

Delete. Removes the last element of the list. Recall from Section 5 that `delete(elem)` and `insert(ref, newElem)` commute when $elem \neq ref$. Since the list is built by successive insertions

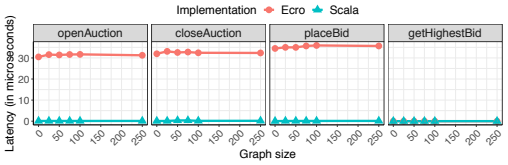


Fig. 6. Latency of RUBiS operations.

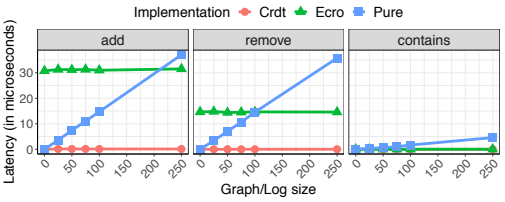


Fig. 7. Latency of operations on add-wins sets for ECROs, CRDTs, and pure-op CRDTs.

and this operation deletes the last element e_n, there are no dependent `insert(`e_n`, _)` operations in the graph. Hence, delete is safe and commutes with all operations from the graph. The ECRO algorithm adds the operation to the graph (O(1)) whereafter it executes the operation (lines 7 to 12 in Algorithm 1). As a result, the relative latency is also approximately 1.

Map. Maps a function over all elements of the list and does not commute with the insert operations that precede it. Algorithm 1 thus adds the operation to the graph (line 7) and adds an hb-edge between every existing `insert` operation and the new `map` operation (lines 8 to 10), before executing the operation (line 12). As a result, the performance of map decreases with the number of non-commutative operations in the graph (i.e. the number of edges that must be added).

The results show that ECROs exhibit latency similar to their sequential implementation for side-effect free operations and commutative operations. For non-commutative operations the latency decreases with the number of non-commutative operations in the graph. The experiment varied the size of the graph from 0 to 100 operations. In practice, the graph will rarely contain as much as 100 operations since the implementation detects causally stable operations and safely removes them from the graph (cf. Algorithm 2). Causal stability was disabled for this experiment in order to study the impact of unstable operations on the algorithm.

RUBiS. The previous list benchmark showcased the worst-case performance of the ECRO algorithm since `map` did not commute with any operation in the execution graph. To get a better understanding of the algorithm's scalability we conduct a similar experiment for RUBiS. We measure the latency of RUBiS operations on an ECRO and compare it to a sequential Scala implementation. In RUBiS, most operations commute, except when they affect the same auction, e.g. `openAuction`, `placeBid`, and `closeAuction`. The `getHighestBid` operation fetches the highest bid for a given auction and thus has no side-effects. We measure the latency for each of these operations on a RUBiS system populated with 100 users, 1000 auctions, and 1000 items. Each operation is executed by a randomly selected user on a random auction.

Figure 6 shows that all operations exhibit *constant* latencies, with a negligible constant time difference between ECROs and Scala for mutating operations, which corresponds to the overhead of the ECRO algorithm. The operations have constant latencies because only a fraction of the operations affect the same auction (i.e. do not commute). Based on this experiment, we conclude that the latency of ECRO operations depends on the number of non-commutative operations in the graph, which in practice is often small, especially if we commit causally stable operations.

6.4.2 Comparison to Existing Set RDTs. We now compare the latency of operations for an add-wins set built atop ECROs with the OR-Set CRDT [Shapiro et al. 2011a] and the pure op-based add-wins set CRDT [Baquero et al. 2017]. Recall that `add(x)` and `remove(y)` commute, except when $x = y$.

We vary the size of the execution graph (for ECROs) and log (for pure-op CRDTs) by executing a random workload before measuring the latency of operations. Figure 7 depicts the results. The latencies remain *constant* for all operations of the add-wins set ECRO and the OR-Set CRDT. The `add` and `remove` operations have a negligible constant time difference between those implementations (less than 0.03ms). In the pure add-wins set CRDT, the latency of operations is linear to the size of the log. This is because the pure operation-based approach checks new operations against all operations in the log in order to compact the log, whereas ECROs only check new operations against non-commutative operations in the graph. When the set's cardinality is big enough and the set operations are distributed uniformly across this space, the number of non-commutative operations is small and thus yields constant latency for ECROs.

6.5 RQ4: Evaluating the Performance of a Geo-Distributed RUBiS Application

Besides the number of non-commutative operations contained by the graph, the performance of the ECRO algorithm also depends on factors such as the load experienced by the system, the latency between replicas, etc. We now compare ECROs with PoR and RedBlue on a geo-distributed RUBiS application by means of a variation on the RUBiS benchmark described by Li et al. [2018].

Setup. The benchmark includes three RUBiS replicas and an independent lock server that coordinates unsafe operations. The RUBiS replicas run on Amazon EC2 `m5.xlarge` virtual machine instances located in three geo-distributed data centers (DCs): Paris, Ohio, and Tokyo. The lock server runs on an `m5.xlarge` instance located in SÃ£o Paulo.

The DC in Paris measures the latency of operations, while the DCs in Ohio and Tokyo execute an update-heavy workload consisting of 100 user requests[11] per second with 50% reads (side-effect free operations, e.g. `getStatus`) and 50% writes (mutating operations, e.g. `openAuction`). Workloads are generated randomly from a probabilistic distribution of the operations. Table 5 shows the latencies and bandwidth between the DCs.

Results. Figure 8 shows the average latency of RUBiS operations as observed by the user at DC Paris. The `getStatus` and `openAuction` operations are safe, hence, they are not coordinated, resulting in low latencies. The `storeBuyNow` and `registerUser` operations are unsafe and require coordination in all implementations (see Table 4), inducing high latencies. Nevertheless, the ECRO implementation reduces latency by more than 10% when compared to PoR and RedBlue. This speedup comes from the fact that ECROs use fine-grained locks on a single user/item, whereas PoR and RedBlue use coarse-grained locks on all users/items thereby preventing any `registerUser`/`storeBuyNow` operations from running concurrently. The `placeBid` and `closeAuction` operations exhibit high latencies for PoR and RedBlue because they are unsafe and require coordination (see Table 4). ECROs do not coordinate these operations because Ordana found a solution to the conflict, which consists of locally ordering `placeBid` operations before concurrent `closeAuction` operations when they affect the same auction (see Table 2 in Section 5). As a result, ECROs achieve low latency (less than 1ms).

We performed the same experiment with a read-mostly workload consisting of 1000 user requests per second with 95% reads and 5% writes. The results are similar and are explained in Appendix C in [De Porre et al. 2021].

[11] In this context a user request corresponds to a method call on a replica.

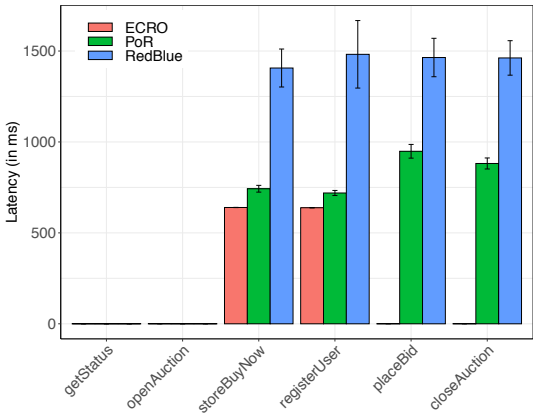


Fig. 8. Average latency of RUBiS operations as observed by users at DC Paris. Error bars represent the 99.9% confidence interval.

Table 5. Average round trip latency and bandwidth between data centers.

Paris	44.3 us 4.78 Gbps			
Ohio	44.4 ms 140 Mbps	66.2 us 4.31 Gbps		
Tokyo	122 ms 49.5 Mbps	79.2 ms 77.9 Mbps	56.2 us 4.75 Gbps	
SÃ£o Paulo	99.3 ms 61.8 Mbps	66.2 ms 97.5 Mbps	135 ms 46.3 Mbps	100 us 4.38 Gbps
	Paris	Ohio	Tokyo	SÃ£o Paulo

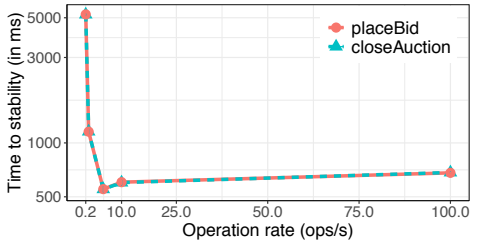


Fig. 9. Time to stability for `placeBid` and `closeAuction` in function of the rate of operations in a geo-distributed RUBiS deployment.

6.6 RQ3: Evaluating the Impact of Causally Unstable Operations on Scalability

As mentioned in Section 6.4, the latency of ECRO operations is related to the number of non-commutative operations in the execution graph. The graph contains tentative operations, i.e. operations that are not yet causally stable[12] and may be reordered due to concurrent operations.

We now turn our attention back to RQ3 and investigate the impact of causally unstable operations on the scalability of the ECRO algorithm. To this end, we measure the time to causal stability using the geo-distributed RUBiS deployment from Section 6.5. Recall that the RUBiS ECRO avoids coordination between the `placeBid` and `closeAuction` operations. However, users may close an auction and concurrently place a bid on that same auction. Upon delivery of the bid, `closeAuction` is reverted, the bid is placed, and the auction is closed again. Hence, there is a time window between closing the auction and declaring the winner, during which new bids may still arrive. Only when `closeAuction` becomes causally stable, the replica declares a winner.

Figure 9 shows the time to stability for the `placeBid` and `closeAuction` operations in function of the rate at which replicas generate operations. The time to stability quickly decreases with the rate of operations because ECROs derive stability from the logical timestamps of incoming operations. If replicas generate an operation every 5 seconds (0.2 ops/s), it takes on average 5 seconds for any operation to stabilize. When replicas generate 5 operations per second, the time to stability decreases to 550ms. Further increasing the rate of operations does not decrease the time to stability due to network latencies, the load experienced by the system, etc.

We conclude that the time to stability is inversely related to the rate at which replicas generate operations. When the rate is high enough (at least a few operations per second), operations stabilize faster than a coordinated execution. Indeed, at a rate of 5 ops/s and more, operations stabilize within 680ms whereas a coordinated execution of `placeBid` or `closeAuction` takes at least 880ms (cf. Fig. 8). In a cloud computing context, we can reasonably assume that data centers are well

[12] An operation is causally stable [Baquero et al. 2017] at a replica when that replica knows that all other replicas observed it.

interconnected and generate operations regularly. Therefore, operations stabilize quickly and the replicas' execution graphs remain reasonably small which yields low latency and good scalability. If some replicas do not generate operations regularly, the time to stability can be reduced by sending acknowledgements for incoming operations, or, by having each replica periodically broadcast its logical clock. For our deployment, replicas could broadcast their clock every 140ms which corresponds roughly to the maximum latency between our DCs (cf. Table 5).

7 RELATED WORK

Inspired by Terry et al. [1995], our work derives correct and efficient RDTs, starting from sequential data types and their distributed semantics, without exposing programmers to merge procedures. Table 6 summarizes how related work guarantees state convergence and preserves application invariants. Existing approaches guarantee state convergence by coordinating non-commutative operations, or by resorting to a stronger consistency model. Our approach (last entry in Table 6) is the only one that does not coordinate non-commutative calls, but instead, deterministically orders them at each replica, in a way that respects causality between dependent operations. Regarding application invariants, existing approaches coordinate unsafe operations to avoid conflicts (i.e. invariant violations) at runtime. In contrast, ECROs allow unsafe operations to run concurrently if a safe reordering of the calls exists. In what follows, we review related work on static analyses, replicated data types, and consistency models.

Static analysis techniques. Our static analysis tool, Ordana, incorporates several static analyses. The first, is a commutativity analysis that detects pairs of non-commutative operations and is similar to well known analyses described by Balegas et al. [2015]; Dimitrov et al. [2014]; Gotsman et al. [2016]; Kulkarni et al. [2011]; Li et al. [2012]. The second, is a dependency analysis that detects dependencies between sequential operations. To this end, it extends the work by Houshmand and Lesani [2019] by taking into account argument relations which enable the detection of fine-grained dependencies between operations. The last, is a safety analysis that detects pairs of operations that may infringe application-level invariants when executed concurrently (similar to [Balegas et al. 2015; Gotsman et al. 2016; Houshmand and Lesani 2019]), and incorporates a novel technique to find solutions without imposing coordination, based on fast local reorderings of conflicting calls. Sivaramakrishnan et al. [2015], Gotsman et al. [2016], and Kaki et al. [2018] statically assign a consistency level to each operation of an RDT. Li et al. [2014] combine static and dynamic analyses to detect invariant-breaking operations and execute them under strong consistency. These approaches may strengthen the consistency level of many operations, thereby, increasing the latency of user requests and deteriorating the scalability and availability of the system. Soethout et al. [2019] statically detect pairs of events that are always independent and thus do not require coordination at runtime. Wang et al. [2019] propose replication-aware linearizability, a criterion that enables sequential reasoning to prove the correctness of CRDT implementations. Nair et al. [2020] focus on the verification of program invariants for state-based replicated objects. These works verify RDTs based on some specification but cannot derive correct RDTs from that specification.

Replicated data types. Conflict-free Replicated Data Types (CRDTs) [Shapiro et al. 2011b] are designed around mathematical properties that guarantee state convergence. These properties impose restrictions on the state or operations (e.g. commutativity) which hampers the design of new CRDTs. Several composition techniques have been proposed by Kleppmann and Beresford [2017]; Meiklejohn and Roy [2015]; Weidner et al. [2020] but none allow arbitrary compositions for all CRDTs. Cloud Types [Burckhardt et al. 2012] do not impose restrictions on operations but require user-defined merge functions. Mergeable Replicated Data Types [Kaki et al. 2019] use a three-way merge function and invertible relational specifications to derive correct merge functions. However,

Table 6. Summary of related approaches.

	Consistency Guarantees	**State Convergence**	**Application Invariants**
CRDTs	SEC	By design	Not supported
Cloud Types	EC	Using user-defined merge procedures	Not supported
Mergeable RDTs	SEC	Using a three-way merge procedure derived from an invertible relational specification of the RDT	Not supported
Indigo	Hybrid	By relying on CRDTs	Coordinate unsafe operations, or, repair broken invariants after the facts
CISE	Hybrid	By relying on CRDTs	Determines consistency models that avoid the conflict
IPA	SEC	By relying on CRDTs	Modify operations' implementation such that conflicts are avoided
RedBlue & PoR	Hybrid	By coordinating non-commutative ops	Coordinate unsafe operations
Quelea & Q9	Hybrid	Picks weakest consistency model that guarantees convergence	Picks weakest consistency model that upholds invariants/contract
Hamsaz & Hampa	Hybrid	By coordinating non-commutative ops	Coordinate unsafe operations
ECROs	Hybrid	By totally ordering non-commutative ops	Reorder unsafe operations and coordinate them if no safe reordering exists

the aforementioned RDTs do not consider application invariants. Hamsaz [Houshmand and Lesani 2019] and Hampa [Li et al. 2020] statically analyze data types to derive coordination protocols that guarantee state convergence and preserve invariants. The derived protocols coordinate all non-commutative and unsafe method calls and are thus more conservative than ECROs. Hampa also provides recency guarantees. Kermarrec et al. [2001] and De Porre et al. [2019] compute a sequential execution of the operations that guarantees convergence and preserves invariants. However, they lack a static analysis to make this efficient. Gallifrey [Milano et al. 2019] proposes orthogonal replication which decouples the conflict resolution strategy of an RDT from its implementation. Gallifrey requires programmers to manually define restrictions over the RDT's operations. In contrast, our approach automatically derives a suitable coordination protocol from the RDT's distributed specification.

Consistency models. Some consistency models [Balegas et al. 2015; Li et al. 2012, 2018; Zhao and Haller 2018, 2020] allow a combination of weak (commutative) operations and strong (coordinated) operations. However, they tend to be conservative as they coordinate all unsafe operations. In contrast, ECROs may avoid coordination by reordering unsafe operations in order not to break invariants, based on the novel combination of static analyses with our execution model. Several programming languages and programming models [De Porre et al. 2020; Holt et al. 2016; Köhler et al. 2020; Milano and Myers 2018; Myter et al. 2018; Zaza and Nystrom 2016] support mixing

consistency levels safely to some extent. However, there is currently no technique that can automatically derive commutative operations (or correct merge procedures) from sequential data types without programmer intervention. Indigo [Balegas et al. 2015] either coordinates unsafe operations or requires programmers to provide a deterministic and monotonic algorithm to repair broken invariants. IPA [Balegas et al. 2018] detects operations that break invariants whereafter programmers must incorporate a suitable conflict resolution and/or coordination technique. While both, Indigo and IPA, start from existing RDTs providing state convergence (e.g. CRDTs) and extend them with invariants, the ECRO approach focuses on deriving those RDTs automatically from a sequential implementation and its accompanying distributed specification.

8 CONCLUSION

We introduce a novel approach to programming geo-distributed applications with Explicitly Consistent Replicated Objects (ECROs). Programmers augment sequential data types with a distributed specification describing the semantics of operations by means of invariants over replicated state. Our static analysis tool Ordana analyzes distributed specifications to detect conflicts, unravel their cause, and find appropriate solutions. This suffices to automatically derive a replicated version of the data type that guarantees convergence and preserves program invariants efficiently.

We presented a portfolio of ECRO data types that demonstrates the flexibility of our approach as it does not require user-defined merge procedures nor data type specific modifications to make operations commute and invariant preserving. Our benchmarks show that ECROs significantly improve the performance of a geo-distributed RUBiS system, when compared to RedBlue and PoR. Unsafe operations are coordinated using a fine-grained locking mechanism that reduces contention. Some conflicts are solved without coordination, reducing latency by several orders of magnitude.

ACKNOWLEDGMENTS

We would like to thank Matteo Marra, Jim Bauwens, and the anonymous reviewers for their comments which helped improve the paper. Kevin De Porre is funded by an SB Fellowship of the Research Foundation - Flanders. Project number: 1S98519N. This work was partially supported by Fundação para a Ciência e a Tecnologia - Portugal (FCT/MCTES) under grants UIDB/04516/2020, PTDC/CCI-INF/32081/2017, and LISBOA-01-0145-FEDER-032662/PTDC/CCI-INF/32662/2017.

REFERENCES

Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. 2015. Efficient State-Based CRDTs by Delta-Mutation. In *Networked Systems - Third International Conference, NETYS 2015, Agadir, Morocco, May 13-15, 2015, Revised Selected Papers (Lecture Notes in Computer Science)*, Vol. 9466. Springer, 62–76. https://doi.org/10.1007/978-3-319-26850-7_5

Peter Bailis, Alan D. Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014. Coordination Avoidance in Database Systems. *Proc. VLDB Endow.* 8, 3 (2014), 185–196. https://doi.org/10.14778/2735508.2735509

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

Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno M. Preguiça, Mahsa Najafzadeh, and Marc Shapiro. 2015. Putting consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*. ACM, 6:1–6:16. https://doi.org/10.1145/2741948.2741972

Carlos Baquero, Paulo S. Almeida, and Ali Shoker. 2017. Pure Operation-Based Replicated Data Types. *CoRR* abs/1710.04469 (2017). arXiv:1710.04469

Eric Brewer. 2012. CAP Twelve years later: How the "Rules" have Changed. *Computer* 45 (02 2012), 23–29. https://doi.org/10.1109/MC.2012.37

Eric A. Brewer. 2000. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 16-19, 2000, Portland, Oregon, USA*. ACM, 7. https://doi.org/10.1145/343477.343502

Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. 2012. Cloud Types for Eventual Consistency. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings (Lecture Notes in Computer Science)*, Vol. 7313. Springer, 283–307. https://doi.org/10.1007/978-3-642-31057-7_14

Emmanuel Cecchet and Julie Marguerite. 2009. RUBiS: Rice University Bidding System. http://rubis.ow2.org/.

Kevin De Porre, Carla Ferreira, Nuno Preguiça, and Elisa Gonzalez Boix. 2021. ECROs: Building Global Scale Systems from Sequential Code (Appendix). http://soft.vub.ac.be/Publications/2021/vub-tr-soft-21-09-appendix.pdf

Kevin De Porre and Elisa Gonzalez Boix. 2019. Squirrel: an extensible distributed key-value store. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection, META@SPLASH 2019, Athens, Greece, October 20, 2019*. ACM, 21–30. https://doi.org/10.1145/3358502.3361271

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 - 19th IFIP WG 6.1 International Conference, DAIS 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17-21, 2019, Proceedings (Lecture Notes in Computer Science)*, Vol. 11534. Springer, 36–56. https://doi.org/10.1007/978-3-030-22496-7_3

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

Dimitar I. Dimitrov, Veselin Raychev, Martin T. Vechev, and Eric Koskinen. 2014. Commutativity race detection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. ACM, 305–315. https://doi.org/10.1145/2594291.2594322

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, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. ACM, 371–384. https://doi.org/10.1145/2837614.2837625

Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. 2016. Incremental Consistency Guarantees for Replicated Objects. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. USENIX Association, 169–184. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/guerraoui

Brandon Holt, James Bornholt, Irene Zhang, Dan R. K. 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, October 5-7, 2016*. ACM, 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 (2019), 74:1–74:32. https://doi.org/10.1145/3290387

Gowtham Kaki, Kapil Earanky, K. C. Sivaramakrishnan, and Suresh Jagannathan. 2018. Safe replication through bounded concurrency verification. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 164:1–164:27. https://doi.org/10.1145/3276534

Gowtham Kaki, Swarn Priya, K. C. Sivaramakrishnan, and Suresh Jagannathan. 2019. Mergeable replicated data types. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 154:1–154:29. https://doi.org/10.1145/3360580

Anne-Marie Kermarrec, Antony I. T. Rowstron, Marc Shapiro, and Peter Druschel. 2001. The IceCube approach to the reconciliation of divergent replicas. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing, PODC 2001, Newport, Rhode Island, USA, August 26-29, 2001*. ACM, 210–218. https://doi.org/10.1145/383962.384020

Martin Kleppmann. 2015. A Critique of the CAP Theorem. *ArXiv* abs/1509.05393 (2015).

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

Mirko Köhler, Nafise Eskandani, Pascal Weisenburger, Alessandro Margara, and Guido Salvaneschi. 2020. Rethinking safe consistency in distributed object-oriented programming. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 188:1–188:30. https://doi.org/10.1145/3428256

Milind Kulkarni, Donald Nguyen, Dimitrios Prountzos, Xin Sui, and Keshav Pingali. 2011. Exploiting the commutativity lattice. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. ACM, 542–555. https://doi.org/10.1145/1993498.1993562

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

Cheng Li, João Leitão, Allen Clement, Nuno M. 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, Philadelphia, PA, USA, June 19-20, 2014*. USENIX Association, 281–292. https://www.usenix.org/conference/atc14/technical-sessions/presentation/li_cheng_2

Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno M. 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 2012, Hollywood, CA, USA, October 8-10, 2012*. USENIX Association, 265–278. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/li

Cheng Li, Nuno M. Preguiça, and Rodrigo Rodrigues. 2018. Fine-grained consistency for geo-replicated systems. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. USENIX Association, 359–372. https://www.usenix.org/conference/atc18/presentation/li-cheng

Xiao Li, Farzin Houshmand, and Mohsen Lesani. 2020. Hampa: Solver-Aided Recency-Aware Replication. In *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I (Lecture Notes in Computer Science)*, Vol. 12224. Springer, 324–349. https://doi.org/10.1007/978-3-030-53288-8_16

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, July 14-16, 2015*. ACM, 184–195. https://doi.org/10.1145/2790449.2790525

Matthew 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, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. ACM, 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, May 16-17, 2019, Providence, RI, USA (LIPIcs)*, Vol. 136. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 11:1–11:19. https://doi.org/10.4230/LIPIcs.SNAPL.2019.11

Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. 2018. A CAPable distributed programming model. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2018, Boston, MA, USA, November 7-8, 2018*. ACM, 88–98. https://doi.org/10.1145/3276954.3276957

Sreeja S. Nair, Gustavo Petri, and Marc Shapiro. 2020. Proving the Safety of Highly-Available Distributed Objects. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science)*, Vol. 12075. Springer, 544–571. https://doi.org/10.1007/978-3-030-44914-8_20

OpenJDK. [n. d.]. JMH - OpenJDK. https://openjdk.java.net/projects/code-tools/jmh/. Accessed: 13-05-2020.

David J. Pearce and Paul H. J. Kelly. 2006. A dynamic topological sort algorithm for directed acyclic graphs. *ACM J. Exp. Algorithmics* 11 (2006). https://doi.org/10.1145/1187436.1210590

Julien Ponge. July 2014. Avoiding Benchmarking Pitfalls on the JVM. https://www.oracle.com/technical-resources/articles/java/architect-benchmarking.html. Accessed: 13-05-2020.

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011a. *A comprehensive study of Convergent and Commutative Replicated Data Types.* Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA. 50 pages.

Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. 2011b. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings (Lecture Notes in Computer Science)*, Vol. 6976. Springer, 386–400. https://doi.org/10.1007/978-3-642-24550-3_29

K. C. Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. 2015. Declarative programming over eventually consistent data stores. (2015), 413–424. https://doi.org/10.1145/2737924.2737981

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, AGERE!@SPLASH 2019, Athens, Greece, October 22, 2019*. ACM, 21–30. https://doi.org/10.1145/3358499.3361222

Douglas B. Terry, Marvin Theimer, Karin Petersen, Alan J. Demers, Mike Spreitzer, and Carl Hauser. 1995. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995*. ACM, 172–183. https://doi.org/10.1145/224056.224070

Werner Vogels. 2009. Eventually Consistent. *Commun. ACM* 52, 1 (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, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. ACM, 980–993. https://doi.org/10.1145/3314221.3314617

Matthew Weidner, Heather Miller, and Christopher Meiklejohn. 2020. Composing and decomposing op-based CRDTs with semidirect products. *Proc. ACM Program. Lang.* 4, ICFP (2020), 94:1–94:27. https://doi.org/10.1145/3408976

Nosheen Zaza and Nathaniel Nystrom. 2016. Data-centric Consistency Policies: A Programming Model for Distributed Applications with Tunable Consistency. In *First Workshop on Programming Models and Languages for Distributed Computing, PMLDC@ECOOP 2016, Rome, Italy, July 17, 2016*. ACM, 3. https://doi.org/10.1145/2957319.2957377

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, AGERE!@SPLASH 2018, Boston, MA, USA, November 5, 2018*. ACM, 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. *J. Log. Algebraic Methods Program.* 114, 100561. https://doi.org/10.1016/j.jlamp.2020.100561