A SCALA DSL FOR FIRST-ORDER LOGIC

The ECRO approach augments sequential data types with a distributed specification. These specifications describe operations and application invariants using first-order logic. As described in Section 2.2, we designed an embedded domain specific language (DSL) for programming first-order logic formulas in Scala, which are then translated to Z3 formulas. We now briefly discuss the different components of the language.

Types, values, and operators. The DSL features three primitive types (booleans, integers, and strings) and supports custom types that can be used to represent complex types such as user-defined classes (cf. Table 1). Primitive values are represented by the `BoolValue`, `IntValue`, and `StringValue` wrappers. We also provide the traditional numeric operators, boolean operators, and comparison operators and provide convenient infix notation for them (cf. Table 2).

Variables and identifiers. Programmers can declare *free variables* by providing a name and type for them and refer to them using *identifiers* (cf. Table 3). Note that declarations do not assign a value to the variable, i.e. they may hold any value of the given type. In order to "assign" a value to a variable, one can state that the variable equals the desired value. For example, `Identifier("age") === IntValue(25)` "assigns" the value 25 to an existing variable "age".

Relations and states. Relations constrain the state of an object. The DSL provides two special state types: `OldState` and `NewState` (cf. Table 3). The former represents the state of the object prior to applying an operation, whereas the latter represents the state after applying the operation. This enables programmers to express the effects of an operation. For instance, the value of a counter can be represented with a relation $value :: State \rightarrow Integer$. Incrementing the counter can then be expressed as `value(newState) === value(oldState) |+| 1`.

Table 1. Types supported by the DSL.

Type	*Type Representation*	*Value Representation*
Boolean	`case object Bool extends Type`	`case class BoolValue(value: Boolean)`
Integer	`case object Integer extends Type`	`case class IntValue(value: Int)`
String	`case object Stringg extends Type`	`case class StringValue(value: String)`
<Custom>	`case class CustomType(name: String) extends Type`	/

Table 2. List of operators provided by the DSL.

Description	*Representation*	*Infix Notation*
Equals	`case class Equals(lhs: Any, rhs: Any)`	`lhs === rhs`
Not Equals	`case class NotEquals(lhs: Any, rhs: Any)`	`lhs <> rhs`
Boolean and	`case class And(lhs: Any, rhs: Any)`	`lhs /\ rhs`
Boolean or	`case class Or(lhs: Any, rhs: Any)`	`lhs \/ rhs`
Negation	`case class Not(stat: Any)`	/
Plus	`case class Plus(lhs: Any, rhs: Any)`	`lhs |+| rhs`
Minus	`case class Minus(lhs: Any, rhs: Any)`	`lhs |-| rhs`
Multiplication	`case class Times(lhs: Any, rhs: Any)`	`lhs |*| rhs`
Division	`case class Divide(lhs: Any, rhs: Any)`	`lhs |/| rhs`
Smaller than	`case class SmallerThan(lhs: Any, rhs: Any)`	`lhs << rhs`
Smaller than or equal to	`case class SmallerThanOrEq(lhs: Any, rhs: Any)`	`lhs <<= rhs`
Bigger than	`case class BiggerThan(lhs: Any, rhs: Any)`	`lhs >> rhs`
Bigger than or equal to	`case class BiggerThanOrEq(lhs: Any, rhs: Any)`	`lhs >>= rhs`

Table 3. Logic building blocks provided by the DSL.

Description	*Representation*	*Notation*
Identifier	`case class Identifier(name: String)`	/
Variable	`case class Var(name: String, tpe: Type)`	`Identifier(name) :: tpe`
Relation	`case class Relation(name: String, vars: Var*)(ret: Type)`	/
First-Order Logic Formula	`case class RelationInstance(name: String, args: Any*)`	`Relation(name, _)(args)`
State	`sealed trait State`	/
Old State	`class OldState extends State`	/
New State	`class NewState extends State`	/
Current State	`class CurrentState extends State`	/
Universal Quantifier	`case class Forall(vars: Set[Var], body: Formula)`	`forall(vars) :- body`
Existential Quantifier	`case class Exists(vars: Set[Var], body: Formula)`	`exists(vars) :- body`
Logical Implication	`case class Implication(ante: Formula, conse: Formula)`	`ante ==> conse`

```scala
case class Relation(name: String, vars: Var*)(ret: Type) {
  def instance(args: Any*): RelationInstance
  def apply(args: Any*) = instance(args:_*)
  def copy(fromTo: (State, State)): Formula
  def copyExcept(fromTo: (State, State), condition: Formula): Formula
  def copyWhen(fromTo: (State, State), condition: Formula): Formula
  // `key`-`v` must be unique
  def unique(key: Var, v: Var, state: State): Formula
  def assertion(cond: Formula, state: State): Formula
}
```

Listing 1. Overview of the relation class' interface.

Custom relations, such as the aforementioned `value` example, are defined by instantiating the `Relation` class shown in listing 1. More concretely, relations are instatiated with a name, a variable number of typed arguments, and a return type. As shown in listing 1, relations provide methods to copy facts from one state to another (`copy`, `copyExcept`, and `copyWhen`), express uniqueness constraints (`unique`), or assert any other condition (`assertion`). Relations are instantiated by applying them to some arguments and yields a first-order logic formula, e.g. `value(newState)`.

Quantifiers and implications. The DSL provides universal and existential quantifiers (`forall` and `exists` functions, cf. Table 3) which take one or more variables and a boolean formula that specifies a property about these variables. Logical implication can be expressed using the ==> infix notation which expects two boolean formulas: the antecedent and the consequent.

A.1 Complete Set Specification

We now present the complete specification of the Set ECROs discussed in Section 2 using our DSL.

Listing 2 shows the distributed specification of the Add-Wins Set ECRO. To represent elements contained by the set we will need a predicate[1] $contains :: V \times State \longrightarrow Boolean$. To this end, line 4 defines a custom type `V` which is the abstract type of the elements that are contained by the set (i.e. it corresponds to the type parameter `V` in `AWSet[V]`). Line 6 then defines the `contains` relation which takes one argument `elem` and is true if `elem` is contained by the set, false otherwise. Note that we do not define a "set" argument explicitly because every relation is defined over a state, hence, the DSL adds a state argument (representing the object) behind the scenes.

[1]A predicate is a relation that returns a boolean.

```
1  case class AWSet[V](set: Set[V]) extends ESet[V]
2  object AWSet extends DistributedSpec {
3    // Declarations
4    val V = CustomType("V")
5    val elem = "elem"
6    val contains = Relation("contains", Var(elem, V))(Bool)
7    val x = Identifier("x")
8
9    // Specs
10   val add = classOf[AWSet[_]].getDeclaredMethod("add", classOf[Object])
11   val remove = classOf[AWSet[_]].getDeclaredMethod("remove", classOf[Object])
12
13   val relations = Set(contains)
14   val operations: Map[Method, Mutator] = Map(
15     add -> Mutator(
16       post = (old: OldState, res: NewState) => {
17         contains(res, x) /\ contains.copyExcept(old -> res, elem === x)
18       },
19       inv = (_: OldState, res: NewState) => contains(res, x)), // add wins
20     remove -> Mutator(
21       post = (old: OldState, res: NewState) => {
22         not (contains(res, x)) /\ contains.copyExcept(old -> res, elem === x)
23       }))
24 }
```

Listing 2. Distributed specification of the Add-Wins Set.

Now that we defined the contains predicate, we can implement the actual specification of the operations. First, we inform the DSL about all relations we will use, by providing a set containing the relations (see the `relations` field on line 13). Then, we provide the DSL with an `operations` field that maps each method to its specification (lines 14 to 23). As explained in Section 2, the postconditions of add and `remove` state that the added element x[2] is present/absent in the resulting set and use the `copyExcept` method defined on relations to copy all the other elements from the `old` set to the `res` set (lines 17 and 22). The invariant on add states that the added element must occur in the resulting state and thus guarantees add-wins semantics. The Remove-Wins Set is similar, except that it puts an invariant on `remove` such that the removed element is not present in the resulting state (cf. Listing 2 in Section 2.2).

A.2 RUBiS Specification

We now present the complete specification of the RUBiS ECRO discussed in Section 2.3 using our DSL. More concretely, we provide the complete implementation of the `placeBid` and `closeAuction` operations for the RUBiS application. Listing 3 shows the sequential implementation of the RUBiS data type. It keeps a set of users and a map from auction IDs to auctions (line 24). Auctions consist of a set of bids, a status (open or closed), and optionally a winner (line 14). Method `placeBid` (line 32) retrieves the auction and places the bid on the auction. `closeAuction` (line 37) retrieves the auction and puts its status on closed.

[2] x is defined on line 7 and corresponds to the parameter of the add and `remove` operations.

To turn this sequential RUBiS data type into an ECRO, we augment it with a distributed specification, shown in Listing 4. First, we declare three first-order logic predicates to represent auctions, users, and bids on auctions: `auction(id, status)`, `user(name)`, and `bid(auction, user, amount)` (line 13 to 15). Then, we use these predicates to describe the `placeBid` and `closeAuction` operations, as explained in Section 2.3. The precondition of `placeBid` (line 32 to 35) requires the auction to be open, the user to exist, the price to be bigger than zero, and every auction to be well-formed (i.e. either open or closed but not both). The postcondition of `placeBid` (line 36 to 37) adds the bid and copies all the existing bids from the old state to the new state. The context of `closeAuction` (line 39) states that the auction was open when the method was called at the origin replica. Its precondition (line 40) states that auctions must be well-formed. Its postcondition (line 41 to 45) closes the auction, states that the auction can no longer be open, and copies all other auctions from the old state to the new state.

```scala
import scala.collection.SortedSet

type User = String
type AID = String
sealed trait Status
case object Open extends Status
case object Closed extends Status

case class Bid(userId: User, bid: Int) extends Ordered[Bid] {
  def compare(that: Bid): Int = bid.compareTo(that.bid)
}
val bidOrdering = Ordering.by[Bid, Bid](b => b.copy(bid = b.bid * -1)) // big to small

case class Auction(bids: SortedSet[Bid] = SortedSet.empty[Bid](bidOrdering), status:
     Status = Open, winner: Option[User] = None) {
  def bid(userId: User, price: Int) = copy(bids = bids + Bid(userId, price))

  def close() = {
    val highestBid: Option[Bid] = bids.headOption
    val winner = highestBid.map(_.userId)
    copy(status = Closed, winner = winner)
  }
}

case class Rubis(users: Set[User] = Set(), auctions: Map[AID, Auction] = Map())
     extends ECRO {
  private def getAuction(auctionId: AID) = {
    auctions.get(auctionId) match {
      case Some(auction) => auction
      case None => throw new Error("Auction " + auctionId + " does not exist.")
    }
  }

  def placeBid(auctionId: AID, userId: User, price: Int): Rubis = {
    val auction = getAuction(auctionId)
    copy(auctions = auctions.updated(auctionId, auction.bid(userId, price)))
  }

  def closeAuction(auctionId: AID): Rubis = {
    val auction = getAuction(auctionId)
    copy(auctions = auctions.updated(auctionId, auction.close))
  }
}
```

Listing 3. Sequential RUBiS implementation.

```
object Rubis extends DistributedSpec {
  // Declarations
  val id = "id"
  val idVar = Variable(id, Stringg)
  val statusV = Variable("status", Bool)
  val auctionVar = Variable("auction", Stringg)
  val userVar = Variable("user", Stringg)
  val amountVar = Variable("amount", Integer)
  val Open = True
  val Closed = False

  // Relations
  val auction = Relation("auction", idVar, statusV)(Bool)
  val user = Relation("user", userVar)(Bool)
  val bid = Relation("bid", auctionVar, userVar, amountVar)(Bool)

  val auctionId = Identifier("auctionId")
  val price = Identifier("price")
  val userId = Identifier("userId")

  val placeBid = classOf[Rubis].getDeclaredMethod("placeBid", classOf[String],
      classOf[String], classOf[Int])
  val closeAuction = classOf[Rubis].getDeclaredMethod("closeAuction", classOf[String])

  // Specs
  val relations = Set(auction, user, bid)

  // auctions are either open or closed but not both
  def auctionsOpenOrClose(state: State) = auction.unique(idVar, statusVar, state)

  val operations: Map[Method, Mutator] = Map(
    placeBid -> Mutator(
      pre = (state: CurrentState) => {
        auction(auctionId, Open, state) /\
        user(userId, state) /\ (price >> 0) /\
        auctionsOpenOrClose(state) }
      post = (old: OldState, res: NewState) => {
        old + bid(auctionId, userId, price, newState) /\ bid.copy(old -> res) }),
    closeAuction -> Mutator(
      ctx = (state: CurrentState) => auction(auctionId, Open, state)
      pre = (state: CurrentState) => auctionsOpenOrClose(state)
      post = (old: OldState, res: NewState) => {
        old + auction(auctionId, Closed, newState) /\
        not (auction(auctionId, Open, newState)) /\
        auction.copyExcept(old -> res, id === auctionId)
      }))
}
```

Listing 4. Distributed specification for RUBiS.

B ECRO REPLICATION ALGORITHM

We now discuss how the ECRO replication algorithm copes with cycles. Afterwards, we provide the complete proof of convergence which was omitted from Section 4.2. To this end, we first introduce some helper lemmas and prove them.

B.1 Detecting and Solving Cycles

We now explain how ECRO's replication algorithm keeps the execution graph acyclic. Algorithm 1 extends the replication algorithm presented in Section 4.1 with a deterministic approach to detect and solve cycles. While adding new edges, the algorithm continuously checks for cycles (line 15). If a newly added edge $c_1 \rightarrow c_2$ causes a cycle, at least one path exists from c_2 to c_1. To solve the cycle, the algorithm computes all paths from c_2 to c_1 (line 38) and breaks them one by one by removing one ao-edge on each path (line 41). These edges can be removed without putting at risk convergence since they impose an artificial ordering between non-commutative operations (say $c_i \xrightarrow{ao} c_j$) but we know that they are already ordered by one or more paths (from c_j to c_i) between them (otherwise they would not be part of the cycle). As a result, we solved the cycle while ensuring that all non-commutative operations remain ordered. Sometimes it is not possible to break each path only by removing ao-edges. In that case the cycle is caused by a combination of hb-edges and co-edges. These cannot be removed as this would violate either convergence or safety. Instead, the algorithm deterministically discards a call that breaks the cycle (line 18). Information about discarded ao-edges and discarded calls is propagated between the replicas to ensure that all replicas eliminate the same ao-edges and/or calls and thus still converge. Since the set of discarded edges and the set of discarded calls grow monotonically and Algorithm 1 is deterministic, all replicas converge to the same execution graph and hence to equivalent states as proven in Section 4.

B.2 Convergence Proof

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

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

Proof. For every local or remote method call c, Algorithm 1 adds c to the replica's execution graph. Therefore, if both replicas observed the same calls, both execution graphs contain the same vertices. We now show that even if the (concurrent) calls were processed in a different order by these replicas, their execution graphs contain the same edges. When a (local or remote) call c is received, Algorithm 1 checks the relation between c and every other call. Hence, independent of the order in which calls are processed, every call is eventually compared to every other call. For every pair of calls $\langle c_1, c_2 \rangle$, the algorithm ensures that both replicas add an hb-edge if $c_1 \prec c_2$ or $c_2 \prec c_1$ and they do not sequentially commute. If c_1 and c_2 are concurrent but do not commute, and Ordana's `resolution` function imposes an ordering between these calls, then both replicas will add the same co-edge. If Ordana does not impose an ordering on these non commutative calls, then both replicas will add the same ao-edge between these calls based on the calls' globally unique identifiers. Therefore, we conclude that both graphs $\mathtt{G}_1$ and $\mathtt{G}_2$ are the same. □

Lemma B.1. *An ECRO replica's execution graph* $\mathtt{G} = \langle \mathtt{C}, \mathtt{E} \rangle$ *contains at most one edge between any two method calls.*

Proof. When a method is called on a replica, it is handled by the `execute_local` function and later integrated at remote replicas using the `execute_remote` function (Algorithm 1). At the origin replica, `execute_local` adds an hb-edge from every previous non-commutative call v in $\mathtt{C}$ to c

Algorithm 1 ECRO replication algorithm: function EXECUTE_REMOTE

```
1: ⟨Σ, σ_0, M, G, t, F⟩, with G = (C, E)                          ▷ ECRO's internal state
2: σ: Σ                                                           ▷ object current state σ
3: discarded                                                      ▷ set of discarded edges
4: function EXECUTE_REMOTE(c)                                     ▷ execution of call c at remote replica
5:     new_edges ← ∅                                              ▷ initialise set to keep edges related to call c
6:     new_discarded ← ∅                                          ▷ initialise set to keep discarded edges related to call c
7:     C ← C ∪ { c }                                              ▷ update graph vertices
8:     E ← E \ discarded                                          ▷ remove discard edges from the following analysis
9:     for v ∈ C ∧ v ≠ c do                                       ▷ determine hb and co-edges between existing calls and call c
10:        if v ≺ c ∧ not seqCommutative(c, v) then
11:            edge ← ⟨v, hb, c⟩                                  ▷ add hb relation for every sequential non-commutative call v
12:        else if v || c then                                    ▷ v is a concurrent call to c
13:            if resolution(c, v) = < then edge ← ⟨c, co, v⟩     ▷ ordering c before v
14:            else if resolution(c, v) = > then edge ← ⟨v, co, c⟩ ▷ ordering v before c
15:        if causesCycle(edge) then                              ▷ checks whether the new edge being added causes a cycle
16:            new_discarded ← resolveCycle(edge)                 ▷ tries to solve cycle by discarding ao-edges
17:            if hasNoSolution() then                            ▷ cycle caused by hb and co-edges
18:                C ← C \ { c }                                  ▷ discard call c
19:                E ← E \ new_edges                              ▷ discard edges related to call c
20:                propagateDiscardedCall(c)                      ▷ propagate discard call to remote replicas
21:                return                                         ▷ function terminates
22:        new_edges ← new_edges ∪ { edge }                       ▷ continue to collect edges related to call c
23:    for v ∈ C ∧ v || c do                                      ▷ determine ao-edges between existing calls and call c
24:        if resolution(c, v) = ⊤ ∧ not commutative(c, v) then
25:            if Id(c) < Id(v) then edge ← ⟨v, ao, c⟩            ▷ arbitrate an order when calls do not commute
26:            else edge ← ⟨v, ao, c⟩
27:            if causesCycle(edge) then                          ▷ an ao-edge that causes a cycle can be immediately discarded
28:                new_discarded ← new_discarded ∪ { edge }       ▷ discard ao-edge that caused a cycle
29:            else new_edges ← new_edges ∪ { edge }              ▷ continue to collect edges related to call c
30:    discarded ← discarded ∪ new_discarded                      ▷ update discarded edges
31:    propagateDiscardedEdges(discarded)                         ▷ if needed, propagate discarded edges to remote replicas
32:    E ← (E ∪ new_edges) \ new_discarded                        ▷ update graph edges
33:    t ← dynamicTopologicalSort(new_edges)
34:    commit()                                                   ▷ Commit causally stable operations
35:    σ ← apply(σ_0, t)                                          ▷ execute the sequence of calls on the initial state σ_0
36: function RESOLVECYCLE(⟨c_1, rel, c_2⟩)
37:    new_discarded ← ∅                                          ▷ initialize set to keep discarded edges to solve the cycle
38:    paths ← allPaths(c_2, c_1, G)                              ▷ determine all paths that close the cycle
39:    for p ∈ paths do
40:        if existsArbitrationOrderEdge(p) then                  ▷ search for ao-edges that are unique to path p
41:            d ← removeEdge(p)                                  ▷ remove the ao-edge that has a minimal id
42:            new_discarded ← new_discarded ∪ { d }              ▷ update discarded edges
43:        else return NO_SOLUTION                                ▷ cycle is caused by an hb or co-edge
44:    return new_discarded
```

(line 10). Hence, there cannot be an hb-edge from c to v or any other type of edge between them. For every incoming call c, `execute_remote` considers two disjoint cases: $v \prec c$ and $v \parallel c$. The case where $c \prec v$ cannot occur because calls are propagated in causal order and we already observed v.

Case 1 ($v \prec c$): if v and c sequentially commute the algorithm does nothing, else, it adds an hb-edge from v to c (line 22). Hence, there cannot be an hb-edge from c to v, nor can there be any other type of edge (co-edge or ao-edge) between v and c since case 1 and 2 are disjoint.

Case 2 ($v \parallel c$): Calls v and c can be safe or unsafe. We thus distinguish two disjoint subcases.

Case 2.1: If v and c are unsafe then we know that Ordana found an ordering of the calls that solves the conflict (either $c < v$ or $v < c$), otherwise Ordana would have restricted the calls and they cannot have executed concurrently (line 6). If `resolution(c, v) = <` then every replica adds a co-edge from c to v (line 25), and there cannot be an edge from v to c, nor can there be any other type of edge between them since case 1 and 2 are disjoint as well as case 2.1 and 2.2. The case where `resolution(c, v) = >` is analogous.

Case 2.2: In this case, calls v and c are safe. If v and c commute the algorithm does nothing, else, it deterministically adds an ao-edge from the call with the smallest ID to the call with the biggest ID (line 28 to 32). Since the identifiers are globally unique, all replicas add the same ao-edge between v and c and there cannot be an edge in the opposite direction. There also cannot be any other type of edge (hb-edge or co-edge) between v and c because case 1 and 2 are disjoint as well as case 2.1 and 2.2.

We thus proved that there can be at most one edge between any two method calls in the graph. □

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. We follow a proof by contradiction. Since both replicas r_1 and r_2 observed the same calls C, 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. Now, assume that their states diverge, i.e. $apply(\sigma_0, \mathtt{t}_1) \not\equiv apply(\sigma_0, \mathtt{t}_2)$. Since r_1 and r_2 diverge we know that at least two non-commutative calls c_1 and c_2 occur in a different order in t_1 and t_2. Let's consider the case where $t_1[c_1] < t_1[c_2]$ and $t_2[c_1] \not< t_2[c_2]$. Since calls c_1 and c_2 do not commute we have to consider three distinct cases. In the first case, calls c_1 and c_2 are unsafe and the algorithm coordinates them to avoid that they execute concurrently (line 6 in Algorithm 1), thus imposing a happened-before relation (hb-edge) between c_1 and c_2 (leading to $t_2[c_1] < t_2[c_2]$). We reach a contradiction since in t_2, by hypothesis, these calls appear in a different order ($t_2[c_1] \not< t_2[c_2]$) but from Lemma B.1 it follows that there is at most one edge between any two calls, i.e. there cannot be an edge from c_2 to c_1 since there is already an hb-edge from c_1 to c_2. In the second case (line 23 to 27 in Algorithm 1), c_1 and c_2 are unsafe but the analysis found a safe ordering of the calls. Assuming the resolution places c_1 before c_2, we again reach a contradiction since in t_2 these calls occur in a different order and there can be at most one edge between them. If the resolution places c_2 before c_1 we reach a similar contradiction because t_1 already has an edge from c_1 to c_2. In the last case (line 28 to 32 in Algorithm 1), calls c_1 and c_2 are safe but do not commute, so the algorithm uses the globally unique identifiers to deterministically order c_1 and c_2 using an ao-edge. Assuming the arbitration relation orders c_1 before c_2, we reach a contradiction since in t_2 these calls occur in a different order and there can be at most one edge between them. If the arbitration relation orders c_2 before c_1, we reach a similar contradiction since in t_1 there is already an edge from c_1 to c_2. The other case where $t_1[c_1] > t_1[c_2]$ and $t_2[c_1] \not> t_2[c_2]$ can be argued likewise.

We showed that both topological orderings t_1 and t_2 keep the relative order of all non-commutative calls. It follows from Definitions 4.2 and 4.3 that the replicas converge. □

C GEO-DISTRIBUTED RUBIS BENCHMARK ON A READ-MOSTLY WORKLOAD

Section 6.5 presented a geo-distributed benchmark for the RUBiS application. The benchmark was executed by measuring the latency of operations at DC Paris while the other DCs execute an update-heavy workload consisting of 100 operations per second with 50% reads and 50% writes. We now perform the same experiment with a read-mostly workload consisting of 1000 operations per second with 95% reads and 5% writes.

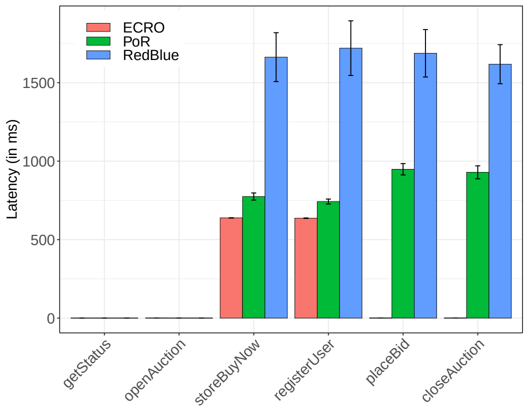

Fig. 1. Average latency of RUBiS operations as observed by users at DC Paris. Error bars represent the 99.9% confidence interval.

Figure 1 shows the average latency of RUBiS operations under this read-mostly workload. The figures are similar to those depicted Section 6.5. 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. The `placeBid` and `closeAuction` operations are unsafe and require coordination in both PoR and RedBlue (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 `closeAuction` operations when they affect the same auction concurrently (see Table 2 in Section 5). As a result, ECROs achieve low latency (less than 1ms) while PoR and RedBlue exhibit high latencies (more than 900ms).